Universidad Nacional de Mar del Plata.

Facultad de Ingeniería.

Departamento de Ingeniería informática.

Asignatura: Taller de programación I.

Trabajo Practico Integrador

Segunda Entrega

Informe

Grupo N°9

Integrantes:

- Gentili, David
- Gomez, Tomas
- Vicente, Juan Martín

Programa desarrollado:

Repositorio:

https://github.com/DavidGentili/tp-taller-de-programacion

Programa a testear

Repositorio:

https://github.com/DavidGentili/codigo-a-testear-tp-taller-de-programacion-I

Testing

Repositorio:

https://github.com/DavidGentili/test-taller-de-promacion-I

INTRODUCCIÓN

El siguiente informe tiene como intención relatar el proceso de testing que se llevó a cabo a partir de un programa brindado por la cátedra. El mismo abarca la aplicación de métodos como son los denominados testeos de caja negra, testeos de caja blanca, pruebas de GUI y pruebas de integración.

Con el fin de poder comprobar los resultados de las operaciones realizadas sobre el modelo, y a sabiendas que el proceso de testing no debe de alterar el código original se tomó la decisión de intervenir el código agregando getters en aquellas clases en las que se consideró que habían quedado pendiente su implementación. Debido a que la ausencia de dichos métodos impedía, en algunos casos, corroborar los datos generados por el constructor, y en otros, directamente no permitía interactuar con el objeto instanciado, evitando así la posibilidad de ser testeado.

Producto: En la búsqueda de poder corroborar la correcta instanciación de un producto, se adiciono el método “getStock”.

Sueldo: Con el fin de poder comprobar la correcta instanciación del elemento sueldo se agrego “getRemuneracionBasica”.

Factura: Para corroborar el correcto funcionamiento de la factura, se generaron los siguientes metodos, “getFecha”, “getPedidos”, “getTotal”, “getFormaPago”, “getPromocionesAplicada”, “getMozo”

Promoción: Con el objetivo de comprobar las correctas instanciaciones de las promociones tanto las de producto como las temporales, se agregaron los métodos “getNombre”, “getFormaPago”, “getPorcentajeDescuento”, “isEsAcumulable”.

Administrador: Con el fin de testear correctamente la implementación de la clase, se decidió adicionar los métodos “getIntancia” y “isEstablecioContrasenia”

Sistema: Con la intención de obtener una mejor forma para corroborar la correcta ABM de los elementos del sistema, se optó por incorporar los métodos “getNombreLocal”, “getMozos”, “getMesas”, “getOperarios”, getAsignacionMesas”, “getPromocionesProducto”, “getPromocionesTemporales”, “getAdministrador”, “getComandas”, “getProductos”.

CAJA NEGRA

Los test de caja negra se caracterizan por ser aquellos en los cuales no se debe conocer el funcionamiento interno del sistema “el como”, si no que se debe conocer las entradas y las salidas de dicho sistema, para así definir clases de equivalencias, y con esto poder elaborar los casos de prueba.

Casos de prueba

En la amplia cantidad de funcionalidades que puede tener un programa, existe la posibilidad que haya métodos en los cuales, debido a su contrato, más específicamente sus precondiciones, no existan clases de equivalencia inválidas, por ende se genere un único caso de prueba. De más está decir que estos suelen ser los menos atractivos, para el análisis de las clases de equivalencias. Por ende se tomó la decisión de no exponer dentro de este informe dichos casos, seleccionando aquellos se creen fundamentales en el flujo del programa o aquellos que son representativos dentro los casos generados, para ser mostrados como ejemplos. Por último, cabe destacar que a este informe se incluyen planillas adjuntas con todos los casos de pruebas realizados.

(Crear imagenes en mejor calidad que el resto de las planillas para mostrar esto)

Promoción Producto - contructor

Nro Escenario	Descripción
1	Hay único escenario posible

Clases validas	Clases invalidas
{"True";"False"} 1	
{"True";"False"} 2	
{"si aplicaDtoPorCantidad es True: dtoPorCantidad_CantMinima>0"} 3	
{"si aplicaDtoPorCantidad es True: dtoPorCantidad_PrecioUnitario>0"} 4	
{"Dia.LUNES";"Dia.MARTES"} 5	

Valores de entrada	Salida esperada
False.	Instancia de PromocionProducto con los valores de entrada como parámetro.
True.	
5	
100	
Dia.JUEVES	
True.	Instancia de PromocionProducto con los valores de entrada como parámetro.
False.	
0	
0	
Dia.JUEVES	

OBSERVACIONES:	A pesar de que ambas son instancias de promocionProducto, se analiza la correcta instanciación de aquellas que aplican descuento por cantidad y dos por uno

Operario - iniciarSesion

CLASE	Operario
METODO	iniciaSesion

ESCENARIOS	
Nro Escenario	Descripcion
1	El operario con el que se quiere ingresar esta activo

TABLA DE PARTICIONES		
Condiciones de entrada	Clases validas	Clases invalidas
Intervalo	Contraseña correcta (1.1)	Contraseña incorrecta (1.2)

BATERIA DE PRUEBAS			
Tipo de Clase	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Escenario 1			
Clase valida	(Contraseña correcta)	Se inicia sesion, y se establece el sistema en modo operario	1.1
Clase invalida	(Contraseña incorrecta)	ContraseniaIncorrectaException	1.2
Escenario 2			
Clase valida	(Contraseña correcta)	UsuarioInactivoException	1.1
Clase invalida	(Contraseña incorrecta)	UsuarioInactivoException | ContraseniaIncorrectaException	1.2

OBSERVACIONES	No hay clases inválidas, debido a que por contrato no existe la posibilidad de ingresar datos incorrectos.

Operario - verificarContrasenia

CLASE	Operario
METODO	verificarContrasenia

ESCENARIOS	
Nro Escenario	Descripcion
1	Hay un único escenario posible

TABLA DE PARTICIONES		
Condiciones de entrada	Clases validas	Clases invalidas
Intervalo	contraseña.length >= 6 && contraseña.length <= 12 (1.1)	contraseña.length < 6 (1.2) || contraseña.length > 12 (1.3)
Intervalo	contraseña contiene un numero (2.1)	contraseña no contiene un numero (2.2)
Intervalo	contraseña contiene mayuscula (3.1)	contraseña no contiene mayuscula (3.2)

BATERIA DE PRUEBAS			
Tipo de Clase	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Clase valida	(Admin1234)	true	1.1, 2.1, 3.1,
Clase invalida	(admin123)	false	3.2
Clase invalida	(adm)	false	1.2
Clase invalida	(Administrador123456789)	false	1.3
Clase invalida	(Adminis)	false	2.2

Sistema - crearComanda

CLASE	Sistema
METODO	crearComanda

ESCENARIOS	
Nro Escenario	Descripción
1	Hay un único escenario

TABLA DE PARTICIONES		
Condiciones de entrada	Clases validas	Clases invalidas
Intervalo	mesa perteneciente a la colección de mesas (1.1)	mesa no perteneciente a la colección de mesas (1.2)
Intervalo	mesa.estaOcupada == false (2.1)	mesa.estaOcupada == true (2.2)

BATERIA DE PRUEBAS			
Tipo de Clase	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Clase valida	(mesa desocupada perteneciente a la colección de mesas)	se ocupa la mesa	1.1, 2.1
Clase invalida	(mesa desocupada no perteneciente a la colección de mesas)	MesaInexistenteException	1.2
Clase invalida	(mesa ocupada perteneciente a la colección de mesas)	MesaOcupadaExceptio n	2.2

Sistema - agregarProducto

CLASE	Sistema
METODO	AgregarProducto

ESCENARIOS	
Nro Escenario	Descripcion
1	El sistema se encuentra en modo administrador
2	El sistema se encuentra en modo operario

TABLA DE PARTICIONES		
Condiciones de entrada	Clases validas	Clases invalidas
Intervalo	Producto no perteneciente a la colección de productos (1.1)	producto perteneciente a la colección de productos (1.2)

BATERIA DE PRUEBAS			
Tipo de Clase	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Escenario 1			
Clase valida	(producto no perteneciente a la colección de productos)	Se agrega el producto a la colección	1.1
Clase invalida	(producto perteneciente a la colección de productos)	ProductoExistenteException	1.2
Escenario 2			
Clase valida	(producto no perteneciente a la colección de productos)	OperacionNoAutorizadaException	1.1
Clase invalida	(producto perteneciente a la colección de productos)	OperacionNoAutorizadaException | ProductoExistenteException	1.2

Sistema - agregarMozo

CLASE	Sistema
METODO	agregarMozo(mozo)

ESCENARIOS	
Nro Escenario	Descripcion
1	El sistema esta inicializado en modo Administrador y con cantidad de mozo menor a la cantidad maxima
2	El sistema esta inicializado en modo Operario y con cantidad de mozo menor a la cantidad maxima
3	El sistema esta inicializado en modo Administrador y con cantidad de mozo mayor o igual a la cantidad maxima
4	El sistema esta inicializado en modo Operario y con cantidad de mozo mayor o igual a la cantidad maxima

TABLA DE PARTICIONES		
Condiciones de entrada	Clases validas	Clases invalidas
Intervalo	mozo que no se encuentra en la colección (1.1)	mozo que se encuentra en la colección (1.2)

BATERIA DE PRUEBAS			
Tipo de Clase	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Escenario 1			
Clase valida	mozo que no se encuentra en la colección	El mozo se agrega a la colección	1.1
Clase invalida	mozo que se encuentra en la colección	MozoExistenteExceptio n	1.2
Escenario 2			
Clase valida	mozo que no se encuentra en la colección	OperacionNoAutorizada Exception	1.1
Clase invalida	mozo que se encuentra en la colección	MozoExistenteExceptio n | OperacionNoAutorizada Exception	1.2
Escenario 3			
Clase valida	mozo que no se	MaximaCantidadMozos	1.1

	encuentra en la colección	Exception	
Clase invalida	mozo que se encuentra en la colección	MozoExistenteExceptio n | MaximaCantidadMozos Exception	1.2
Escenario 4			
Clase valida	mozo que no se encuentra en la colección	OperacionNoAutorizada | MaximaCantidadMozos Exception	1.1
Clase invalida	mozo que se encuentra en la colección	OperacionNoAutorizada | MaximaCantidadMozos Exception | MozoExistenteExceptio n	1.2

Testeos

Una vez documentadas las clases de equivalencias, y sus casos de prueba, se codificaron los testeos correspondientes, se elaboraron unos 87 test, de los cuales solo 7 tuvieron una ejecución exitosa, el resto de los test tuvieron conflictos para terminar, en su gran mayoría, inconsistencias en los asertos de postcondiciones, en su mayoría bajo el error “java.lang.AssertionError: No se creó la instancia del administrador”, pero otros mostraron el error “java.lang.AssertionError: La capacidad de la barra debe ser al menos 1”.

Ante esta situación, se hace notable la imposibilidad de ejecutar pruebas de cobertura realistas, debido a que con esta condición solo se ejecutaban un 25% de los métodos y tan solo un 19% de las líneas de código, haciendo casi irrisorio la realización de pruebas de caja blanca.

Estos resultados llevaron al grupo a realizar pruebas estáticas tales como inspecciones, ubicando la mayor cantidad de los conflictos en la inicialización del Sistema y del Administrador. para ser más específicos se encontró que en la clase Administrador, en el método, “crearAdministrador”, se impone como postcondición la asignación de una nueva instancia de Administrador a la variable estática “instancia”, y dicha condición no se cumple.

Así también se encontró que en el método “inicializarSistema”, de la clase Sistema, al final de la inicialización, invoca al invariante, el cual obliga al sistema a poseer al menos dos productos en promoción, dicha condición no es factible, debido a que no se posee una instancia del sistema, de hecho se inicializa con la intención de adquirir dicha referencia. por ende tal inicialización termina, mediante un aserto, impidiendo la manipulación del sistema.

Por último, se encontró que el constructor de la clase Mesa, también posee una precondición que impide la interacción con la referencia de la clase, debido a que solicita en dos precondiciones distintas que el número de la mesa tiene que ser igual a 0 y la capacidad mayor o igual a 1 y así también solicita que el número de mesa tiene que ser mayor o igual a 1 y la capacidad debe ser mayor o igual a 2. Siendo imposible que el número de mesas sea igual a 0 y mayor o igual a 1, impidiendo que la clase sea instanciada.

Se retoma nuevamente, la idea de que el testing solo debería de retornar al equipo de desarrollo el reporte con los errores encontrados, y no modificar el código, pero siendo que la magnitud de los errores impide la posibilidad siquiera de comenzar a ejecutar dichas pruebas, se decide realizar las siguientes modificaciones puntuales en los asertos del código, adaptandolos a los contratos estipulados en el javadoc y en el documento de especificación de requerimientos entregado por la cátedra.

Administrador - crearAdministrador

```
public static Administrador crearAdministrador() throws AdministradorExistenteException {
    if (inicializado)
        throw new AdministradorExistenteException();

    Administrador administrador = new Administrador();
    inicializado = true;
    instancia = administrador;

    assert instancia != null : "No se creó la instancia del administrador";
    return administrador;
}
```

Mesa - Constructor e invariante

```java
public Mesa(int nroMesa, int capacidad) {
    assert nroMesa >= 0 : "El número de mesa no puede ser negativo";
    assert capacidad > 0 : "La capacidad no puede ser negativa";
    assert nroMesa == 0 ? capacidad >= 1 : capacidad >= 2 : "debe respetar la capacidad de la barra/mesa";

    this.nroMesa = nroMesa;
    this.capacidad = capacidad;
    this.estaOcupada = false;

    assert this.nroMesa == nroMesa : "El número de mesa no se ha asignado correctamente";
    assert this.capacidad == capacidad : "La capacidad no se ha asignado correctamente";
    verificarInvariantes();
}
```

```java
private void verificarInvariantes() {
    assert nroMesa >= 0 : "El número de mesa no puede ser negativo";
    assert capacidad > 0 : "La capacidad no puede ser negativa";

    assert nroMesa == 0 ? capacidad >= 1 : capacidad >= 2 : "debe respetar la capacidad de la barra/mesa";

}
```

Sistema - Invariante

En este caso solo se decidió comentar el aserto que limita la cantidad de productos en promoción, debido a que esto es condición para realizar pedidos e interactuar con comandas, pero no es invariante de toda la ejecución del sistema.

Con estas 4 modificaciones, loga que de los 87 test el 72 logre resultado exitoso, consiguiendo así que se recorran un 89% de los métodos del modelo y un 88% de las líneas de código del modelo, siendo estos, resultados más factibles que permitan la evolución del resto de los tests.

Errores registrado de los testeos

A Partir de las modificaciones y la nueva ejecución de los test de caja negra, se consiguió elaborar una planilla de errores, donde pudimos registrar las clases y los metodos que tuvieron errores, ademas de cual fue la prueba que produjo dicho error, y por último una leve interpretación de cuáles los motivos de que la prueba “fracasara”.

Pruebas de cobertura

Una vez realizado los test, y con los resultados de las pruebas de cobertura, se almacenan dichos datos en un reporte que almacena el porcentaje de clases recorridas, el porcentaje de métodos ejecutados, y el porcentaje de líneas de código ejecutadas. Con estos datos pudimos comenzar a preparar un nuevo conjunto de datos que integrarán las pruebas de caja blanca.

PRUEBA DE CAJA BLANCA

Análisis de los resultados de cobertura

Realizados los Test de Cobertura nos encontramos que no recorrimos todas las líneas código, en su gran mayoría resultan ser métodos como getters, pero también existen “caminos” sin recorrer en métodos más atractivos de analizar como los siguientes:

ID	CLASE	MÉTODO
1	Sistema	inicializarSistema
2	Sistema	eliminarProducto
3	Sistema	asignarMesa
4	Sistema	cerrarComanda
5	Sistema	buscaOperario

Analizando cada uno de ellos nos encontramos lo siguiente:

1 - inicializarSistema

Nunca vamos a pasar por la excepción de Administrador Existente, debido a que si se creó previamente un Administrador, automáticamente se generará una instancia de Sistema, por lo que éste método arrojaría la excepción de Sistema Ya Inicializado.

2 - eliminarProducto

Esto se debe a que dentro del contrato, no se especifica como precondición que el producto se encuentre en el sistema, pero hay un aserto que lo limita, así también, hay una excepción que controla este error.

3 - asignarMesa

Se encuentra un error en el condicional que analiza si la mesa está incluida en la colección de mesas, no referencia a la mesa ingresada por parámetro, si no a la misma colección de mesas.

4 - cerrarComanda

Para este método se decidió realizar el Test de Caja Blanca

5 - buscaOperario

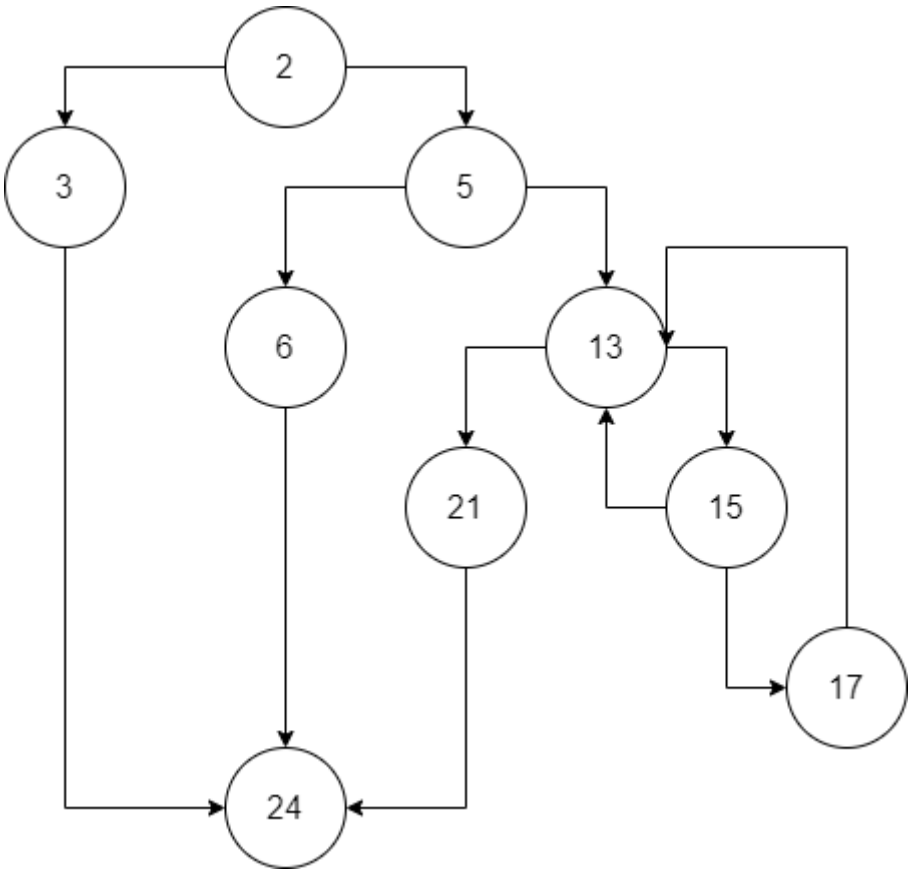
Para este método se decidió realizar el Test de Caja Blanca

Test de Caja Blanca - Grafo ciclomático

cerrarComanda

```
  public void cerrarComanda(Mesa mesa) throws MesaInexistenteException, MesaNoOcupadaException {
      assert mesa != null : "La mesa no puede ser nula";
1
2      if (!mesas.contains(mesa))
3         throw new MesaInexistenteException(mesa);
4
5      if (!mesa.estaOcupada()) {
6           throw new MesaNoOcupadaException(mesa);
7       }
8
9       Comanda comanda = comandas.get(mesa);
10      Mozo mozo = null;
11      boolean encontrado = false;
12      Iterator<Map.Entry<Mozo, List<Mesa>>> it = asignacionMesas.entrySet().iterator();
13      while (it.hasNext() && !encontrado) {
14          Map.Entry<Mozo, List<Mesa>> entry = it.next();
15          if (entry.getValue().contains(mesa)) {
16              mozo = entry.getKey();
17              encontrado = true;
18          }
19      }
20
21      comandas.remove(mesa);
22      mesa.desocupar();
23
24       // TODO: Hacer facturación con la comanda.

      assert !comandas.containsKey(mesa) : "La comanda no se cerró";
      verificarInvariantes();
  }
```



Complejidad Ciclomática: 5

Posibles caminos:

N° camino	Camino
1	2 - 3 - 24
2	2- 5 - 6 - 24
3	2 - 5 - 13 - 21 - 24
4	2 - 5 - 13 - 15 - 13 - 21 - 24
5	2 - 5 - 13 - 15 - 17 - 13 - 21 - 24

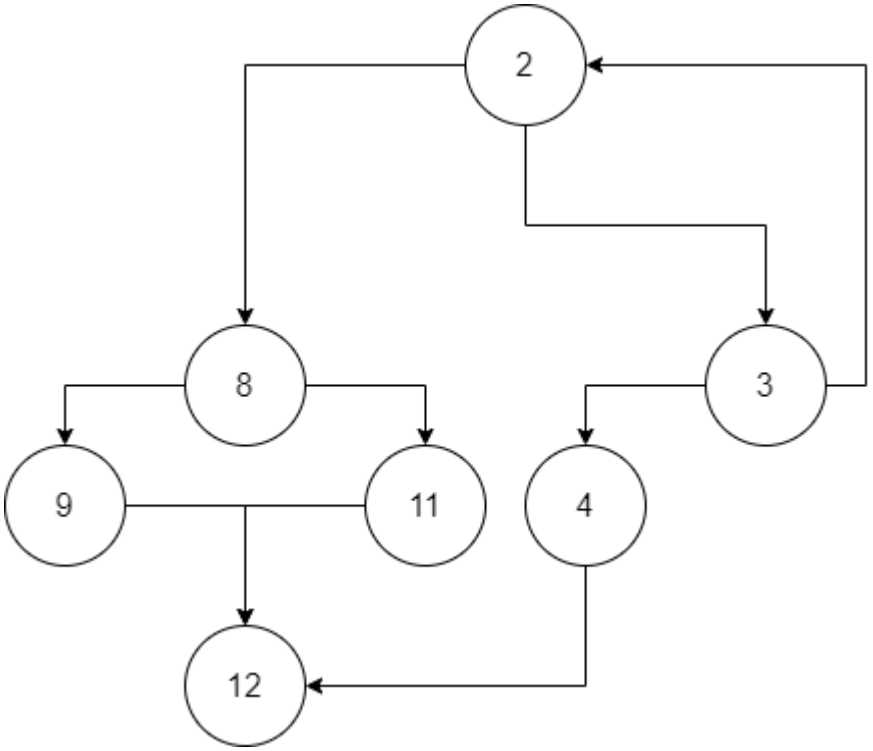
Camino	Parámetros de Entrada	Resultado Esperado
1	mesa no incluída en la lista de mesas del sistema.	Lanza la excepción correspondiente.
2	mesa incluída en la lista de mesas del sistema que no se encuentra ocupada.	Lanza la excepción correspondiente.
3	mesa incluída en la lista de mesas del sistema que se encuentra ocupada. Sin asignaciones en la colección de asignaciones	cierra la comanda, no se cuenta con el mozo para la factura
4	mesa incluída en la lista de mesas del sistema que se encuentra ocupada. mozo en la colección de asignaciones, pero sin mesas asignadas	cierra la comanda, no se cuenta con el mozo para la factura
5	mesa incluída en la lista de mesas del sistema que se encuentra ocupada. mozo en la colección de asignaciones, con la mesa correspondiente asignada	cierra la comanda, se obtiene el mozo que atendió la mesa

Una vez identificados los posibles caminos, se identificaron aquellos que no fueron explorados en las pruebas de caja negra, en este caso, quedaría realizar los test correspondientes al camino 4 y 5, desde los métodos que posee la clase Sistema no es posible, generar un escenario factible con el camino 4, debido a que al incorporar una mesa y un mozo a la colección de asignaciones, automáticamente, se estaría instanciando un mozo con al menos una mesa. Por tal motivo, solo es posible realizar el camino 5, que corresponde a asignar un mozo a la mesa en la cual se está cerrando la comanda, esto se vera reflejado en el test “testCerrarComandaMesaEnListaDeAsignaciones”.

buscarOperario

```
1   public Operario buscarOperario(String nombreUsuario) throws OperarioInexistenteException {
2       for (Operario operario : operarios) {
3           if (operario.getNombreUsuario().equals(nombreUsuario)) {
4               return operario;
5           }
6       }
7
```

```
8       if (administrador.getNombreUsuario().equals(nombreUsuario)) {
9           return administrador;
10      } else {
11          throw new OperarioInexistenteException(nombreUsuario);
12      }
13  }
```



Complejidad Ciclomática: 3

Posibles caminos:

N° camino	Camino
1	2 - 8 - 9 - 12
2	2- 8 - 11 - 12
3	2 - 3 - 2 - 8 - 9 - 12
4	2 - 3 - 4 - 12

Camino	Parámetros de Entrada	Resultado Esperado
1	nombre de usuario del administrador, sin operarios almacenados	administrador
2	nombre de usuario no correspondiente con ningún operario	OperarioInexistenteException
3	nombre de usuario del administrador, con operarios almacenados	administrador
4	nombre de usuario de un operario	operario

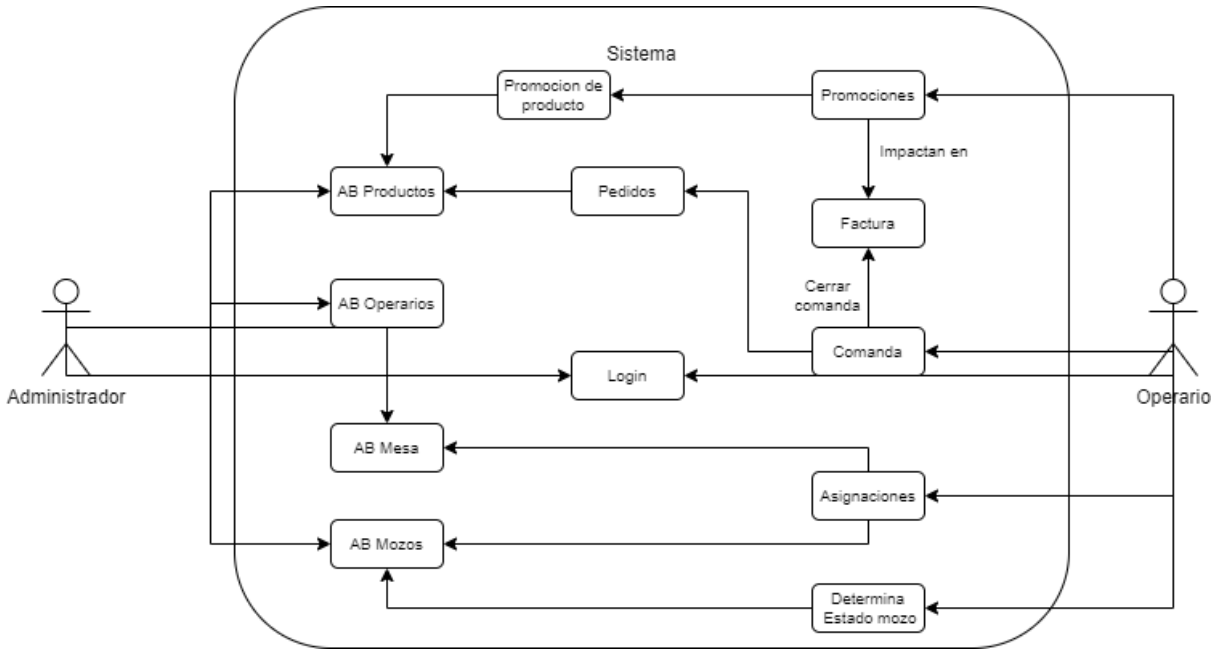
Con los posibles caminos identificados, podemos determinar que solo hay uno de ellos que no ha sido probado en caja negra, este es el camino 3. por ende se prepara un test denominado “testBuscarOperarioAdministrador”

Análisis de resultados de caja blanca

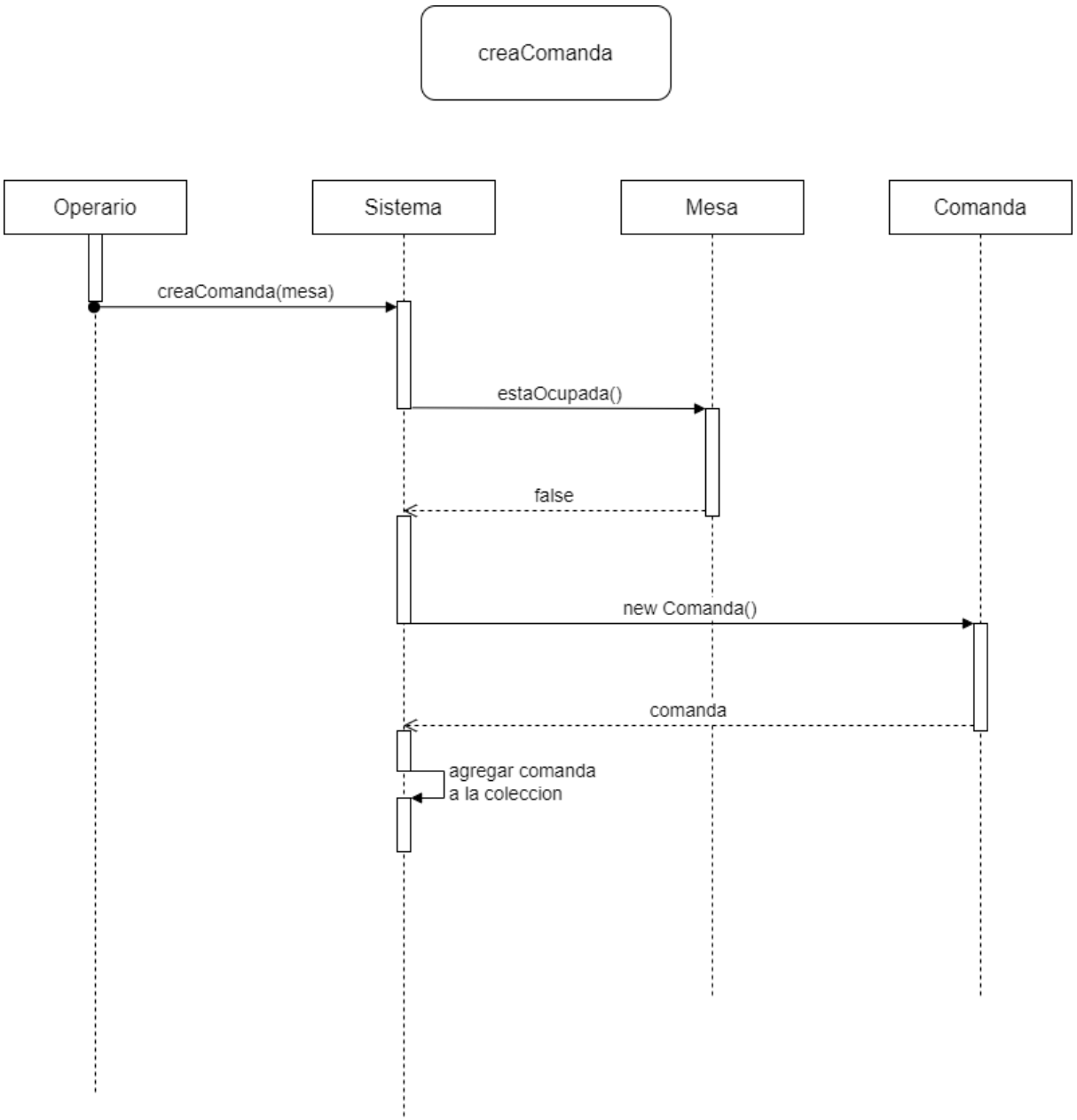
Al ejecutar los dos test que se crearon, el test de “testBuscarOperarioAdministrador” no tuvo ningún conflicto, su ejecución fue “exitosa”, en cambio, “testCerrarComandaMesaEnListaDeAsignaciones”, mostro nuevamente un error que ya se había dictaminado en los testeos de caja negra, el mismo determina que el método “asignarMesa” tiene una comparación incorrecta, por ende no se pueden realizar las asignaciones correspondientes.

PRUEBAS DE INTEGRACIÓN

Para realizar un ejemplo de la prueba de integración, se decidió diseñar un diagrama de caso de uso, que representara la interacción que poseen los dos tipos de “usuario”, para con los elementos del sistema.



A partir de dicho caso de uso se diseñó un ejemplo de un caso de prueba correspondiente a la creación de una comanda. Además, se elaboró el diagrama de secuencias correspondientes para mayor claridad.



Este caso de prueba determina dos escenarios posibles, en el primero el sistema se encuentra inicializado con la mesa dentro de la colección de mesas, y la segunda no contiene agregada a la mesa en la colección.

A su vez, la mesa puede encontrarse en estado ocupada y en estado liberada, esto nos permite determinar las siguientes combinaciones de entradas salidas

N° de escenario	Descripción
1	El sistema se encuentra inicializado con la mesa en su colección
2	El sistema se encuentra inicializado con sus colecciones vacías

N° de caso	Entrada	Salida esperada
1	mesa ocupada Escenario 1	MesaOcupadaException
2	mesa desocupada Escenario 1	se agrega una nueva comanda a la colección
3	mesa ocupada Escenario 2	MesaIntexistenteExceptio n
4	mesa desocupada Escenario 2	MesaIntexistenteExceptio n

CONCLUSIÓN

La realización de este trabajo nos permitió tomar un fuerte contacto práctico con varias técnicas del desarrollo de software que intentan mejorar la calidad del producto final. Brindándonos un entendimiento general de la interacción que posee un equipo de desarrollo y un equipo de control de calidad, permitiéndonos adoptar ambos roles en el transcurso del trabajo.

Desde el lado del desarrollo, nos permitió poner en práctica conceptos vistos en el transcurso de la materia, como el proceso de desarrollo disciplinado, y su importancia para generar un software robusto y de calidad.

Con respecto al testing, se implementaron levemente procesos de pruebas estáticas, con el fin de comprender el funcionamiento de la pieza de software brindada para testear. a partir de dicho proceso, se pudo diseñar un plan de pruebas dinámicas, que permitió desarrollar el testeo de varios módulos en simultáneo, generando así, varias baterías o “sets” o “suits” de pruebas de caja negra.

Una vez terminado de explorar los conceptos de pruebas de caja negra, se nos presentó la posibilidad, de implementar análisis de cobertura con dichos testeos, y desarrollar nuevos casos de pruebas de caja blanca, con el fin de ver como aun así cubriendo todas las clases de equivalencias, no se habían cubierto todos los posibles test.

Por último, dicho trabajo, nos brindó la posibilidad de interactuar con herramientas sumamente útiles a futuro en nuestra vida profesional como integrantes dentro del proceso de desarrollo del software.