Trabajo Práctico Especial
Sistema de Cervecería
Grupo 5

Asignatura: Taller de Programación I

Cátedra

Guccione, Leonel

Lazzurri, Guillermo

Remón, Cristian

Spinelli, Adolfo Tomás

Subgrupo tester

Ávalos, Wenceslao

Lapiana, Santiago Nicolás

Subgrupo cuyo código fue testeado

Luna, Lautaro

Sosa, Santiago

Video

https://www.youtube.com/watch?v=Kbid5r0Fm70

Repositorio

https://github.com/santilapi13/taller-de-programacion-1

Índice

1. Introducción

En este informe se busca detallar el proceso de testing de un sistema de gestión de mesas, personal y ventas de una cervecería. En el mismo, se exploraron distintos tipos de test, desde pruebas unitarias hasta pruebas de integración. Para profundizar lo dicho, se definirán los distintos tipos de tests:

- **Test de Caja Negra:** En él, se busca analizar las salidas de un módulo (en el caso actual dentro del paradigma orientado a objetos, un método) a partir de un conjunto de entradas, sin tener la información de cómo se llevan a cabo las distintas operaciones.
- **Test de Caja Blanca:** A diferencia de caja negra, busca analizar los distintos flujos (hablando ya del código que lo compone) que pueden darse en un mismo módulo según un conjunto diverso de entradas.
- **Test de Persistencia:** Tiene por finalidad comparar la información que se persiste en memoria secundaria con la que luego se carga en memoria al despersistir, para determinar si este proceso se realiza satisfactoriamente.
- **Test de GUI:** Consiste en revisar el accionar de la interfaz gráfica del sistema para que la comunicación con el usuario sea lo más amena posible y realice las validaciones que deba realizar.
- **Test de Integración:** Deja de lado los tests unitarios para analizar la interacción entre distintos módulos y ver cómo se integran para llevar a cabo procesos más complejos.

2. Test de Caja Negra

El test de Caja Negra es un tipo de test con la particularidad de que no se posee acceso al código. Por ende estas se centran en lo que se espera de un módulo, y el test se basa en suministrar datos de entrada y analizar el comportamiento de la salida.

Para realizarlas se necesita contar con la especificación de requerimientos de cada módulo a testear, la cual fue brindada por el subgrupo cuyo programa fue testeado.

Primero que nada se crearon los casos de prueba de los métodos que estaban documentados con sus precondiciones, postcondiciones y las excepciones que pueden lanzar. Además se incluyeron otros métodos de los que tal vez faltaba una más clara especificacion pero que entendemos son necesarios para analizar completamente el programa.

Todo lo mencionado se encuentra documentado en el pdf.

"Caja Negra documentado - Avalos Wenceslao y Santiago Lapiana - Subgrupo 5."

En este documento se encuentran todos los métodos testeados, con una información clara de su procedimiento, incluyendo los escenarios, las tablas de particiones y la batería de pruebas.

Además, para testear los métodos del programa en Java, se utilizaron las dependencias de JUnit, los tests estan en el paquete

"testCajaNegra"

Y los escenarios para realizarlos se encuentran en el paquete.

"escenarios"

Con la herramienta JUnit se procedió a plasmar todo lo documentado en código y se realizaron en total *47 tests de 16 metodos*. Del total mencionado hubo 3 casos de prueba con fallos.

Si bien todo el programa en sí funciona muy bien, cumpliendo casi siempre con lo esperado. Se notó una tendencia en los 3 fallos indicados en la implementación de las promociones y los productos que incluyen. Por ejemplo, está especificado y validado mediante una excepción que no se puede eliminar un producto que está incluido en una comanda abierta, y el test de tal acción (realizado en EliminarProductoEnComandaAbierta()) fue fallido. De la misma

forma, al aplicar una promoción de un producto se obtuvo otro fallo y también al querer agregar una nueva promoción repetida de un producto específico.

Los fallos entonces serían los siguientes, siendo el *más importante el de la aplicación de una promoción de producto a la venta final*:

-EliminarProductoEnComandaAbierta → Se puede eliminar un producto que esta en una comanda abierta.
-AgregarPromoProductoExistenteConProductoListado → No se detecta al agregar una promoción de producto repetida, que ya fue creada.
-CerrarMesaValidaConPromocionProducto → La promoción del producto no se aplica correctamente.

En resumen, en el paquete de java se pueden encontrar todos los test realizados de caja negra, y los pasos para realizarlos están en el pdf mencionado más arriba.

3. Test de Caja Blanca

Los test de Caja Blanca se centran en los detalles del procedimiento del software, por ende, para realizarlas se necesita del código fuente. Éstas, generalmente se aplican a las unidades de software, y el objetivo al aplicarlas es diseñar casos de pruebas para que se ejecuten al menos una vez todas las sentencias del programa, y todas las condiciones por cualquiera de sus posibles ramas.

Para determinar a que unidad aplicar este test, se realizó una prueba de cobertura del programa(Utilizando la herramienta eClemma que fue dada a conocer por la cátedra) y se noto que el metodo

"AsignarMesa(Mozo mozo, Mesa mesa"

Posee poca cobertura, como se puede ver en la imagen a continuación.

COBERTURA INICIAL:

```
public void asignarMesa(Mozo mozo, Mesa mesa) throws MozoNoDisponibleException, MozoInexistenteException,
        MesaNoDisponibleException, MesaInexistenteException {

    int i = 0;

    while (i < mozos.size() && !mozo.equals(mozos.get(i)))
        i++;

    if (i < mozos.size() && mozo.equals(mozos.get(i))) { // existe el mozo

        if (mozo.getEstado() == Estado.ACTIVO) { // el mozo esta disponible
            i = 0;

            while (i < mesas.size() && mesa.getNroMesa()!=mesas.get(i).getNroMesa())
                i++;

            if (i < mesas.size() && mesa.getNroMesa()==mesas.get(i).getNroMesa()) { // la mesa existe

                if (mesa.getEstado().equalsIgnoreCase("LIBRE") && !mesa.isAsignado()) { // el mozo y la mesa estan disponibles
                    mozo.addMesa(mesa);
                    mesa.setAsignado(true);

                }
                else { //la mesa no esta disponible
                    String msg;
                    if (mesa.isAsignado())
                         msg = "ERROR: La mesa ya estaba asignada";
                    else
                        msg="La mesa " + mesa.getNroMesa() + " no esta disponible";
                    throw new MesaNoDisponibleException(msg);
                }

            } else
                throw new MesaInexistenteException("La mesa seleccionada no existe");
        } else
            throw new MozoNoDisponibleException("El mozo " + mozo.getNya() + " no esta disponible");
    } else
        throw new MozoInexistenteException("El mozo seleccionado no existe");

    this.invariante();
}
```

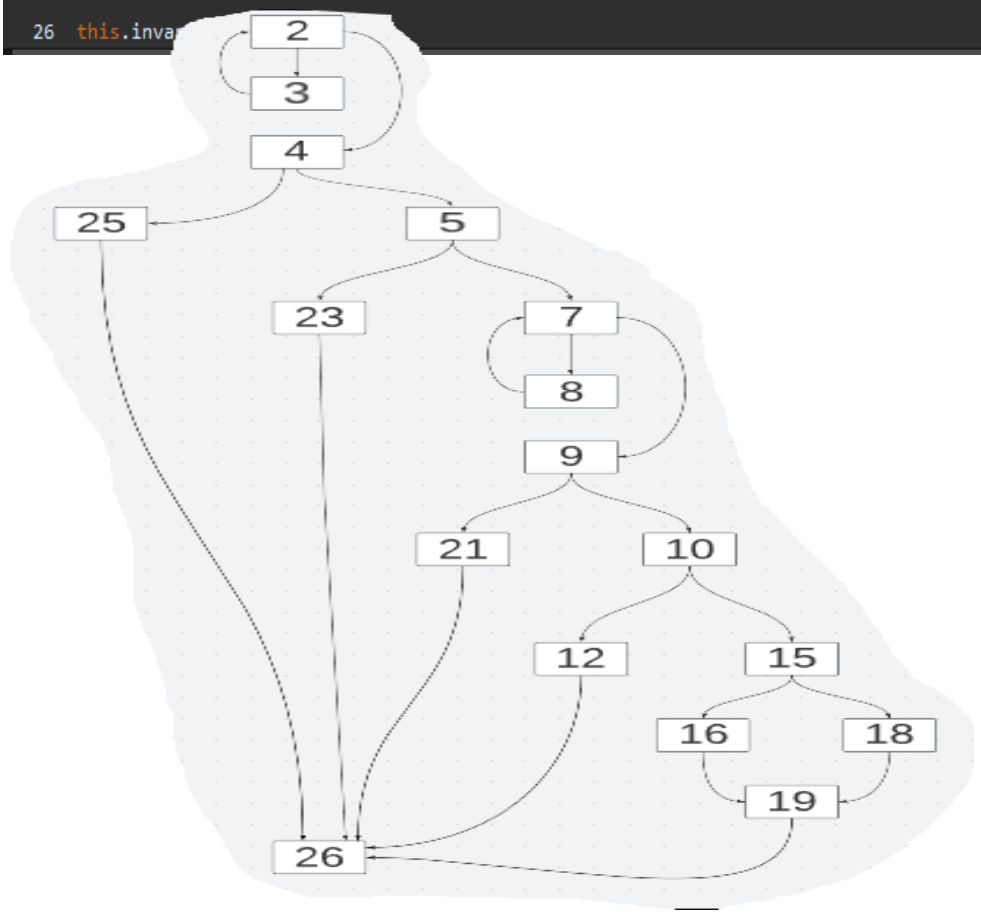
En base a esto concluimos en realizar un test de caja blanca de dicho método, aprovechando que posee varios condicionales en él.

Para realizar el test, primero numeramos cada línea del método y luego creamos el grafo ciclomático asociado a él. Se puede ver en la siguiente página del informe.

Luego procedimos a calcular su complejidad ciclomática y entonces creamos los caminos con sus respectivos casos de prueba para poder realizar una cobertura total de la unidad.

Podemos destacar una dificultad que nos encontramos en una línea particular, la cual se explicará más adelante.

```
public void asignarMesa(Mozo mozo, Mesa mesa) throws MozoNoDisponibleException, MozoInexistenteExcept
1   int i = 0;
2   while (i < mozos.size() && !mozo.equals(mozos.get(i)))
3       i++;
4   if (i < mozos.size() && mozo.equals(mozos.get(i))) { // existe el mozo
5       if (mozo.getEstado() == Estado.ACTIVO) { // el mozo esta disponible
6           i = 0;
7           while (i < mesas.size() && mesa.getNroMesa()!=mesas.get(i).getNroMesa())
8               i++;
9           if (i < mesas.size() && mesa.getNroMesa()==mesas.get(i).getNroMesa()) { // la mesa existe
10              if (mesa.getEstado().equalsIgnoreCase("LIBRE") && !mesa.isAsignado()) { // el mozo y
11                  mozo.addMesa(mesa);
12                  mesa.setAsignado(true);
                }
13              else { //la mesa no esta disponible
14                  String msg;
15                  if (mesa.isAsignado())
16                      msg = "ERROR: La mesa ya estaba asignada";
17                  else
18                      msg="La mesa " + mesa.getNroMesa() + " no esta disponible";
19                  throw new MesaNoDisponibleException(msg);
                }
20          } else
21              throw new MesaInexistenteException("La mesa seleccionada no existe");
22      } else
23          throw new MozoNoDisponibleException("El mozo " + mozo.getNya() + " no esta disponible");
24  } else
25      throw new MozoInexistenteException("El mozo seleccionado no existe");
26  this.inva
```



La condición de la línea numerada "15" restringe el paso del hilo de ejecución por la línea "18" dado para que una mesa esté ocupada, inevitablemente antes fue asignada a un mozo. Por eso, dicha línea de código es *inalcanzable* y no fue considerada por ninguno de los caminos.

Complejidad:

- FORMA 1:
  V(G) = 23 arcos - 17 nodos + 2 = **8**
- FORMA 2:
  V(G) = 7 condicionales + 1 = **8**
- FORMA 3:
  V(G) = 7 regiones cerradas + exterior = **8**.

Caminos de cobertura (nro de caminos <= complejidad)**:**

Nro	Camino
1	2-4-25-26
2	2-3-2-4-25-26
3	2-3-2-4-5-23-26
4	2-3-2-4-5-7-9-21-26
5	2-3-2-4-5-7-8-7-9-10-12-26
6	2-3-2-4-5-7-8-7-8-7-9-10-15-16-19-26

Escenarios:

Escenario	Descripción
1	No hay mesas ni mozos en el sistema.
2	No hay mesas, pero hay un mozo de franco en el sistema.
3	No hay mesas y únicamente un mozo activo.
4	Hay una mesa no asignada y un mozo activo.
5	Único mozo activo, mesa 1 sin asignar y mesa 2 asignada a él.

Casos de prueba:

Camino	Entrada	Salida esperada	Escenario
1	Mesa = mesa cualquiera. Mozo = mozo cualquiera.	MozoInexistenteException()	1
2	Mesa = mesa cualquiera. Mozo = mozo distinto al único que existe.	MozoInexistenteException()	2
3	Mesa = mesa cualquiera. Mozo = el único mozo del sistema.	MozoNoDisponibleException()	2
4	Mesa = mesa cualquiera. Mozo = único mozo del sistema.	MesaInexistenteException()	3
5	Mesa = única mesa del sistema. Mozo = único mozo del sistema.	Se asigna el mozo a la mesa exitosamente.	4
6	Mesa = mesa 2. Mozo = único mozo del sistema.	MesaNoDisponibleException() porque la mesa ya fue asignada.	5

Cobertura final del método:

```
public void asignarMesa(Mozo mozo, Mesa mesa) throws MozoNoDisponibleException, MozoInexistenteException,
        MesaNoDisponibleException, MesaInexistenteException {

    int i = 0;

    while (i < mozos.size() && !mozo.equals(mozos.get(i)))
        i++;

    if (i < mozos.size() && mozo.equals(mozos.get(i))) { // existe el mozo

        if (mozo.getEstado() == Estado.ACTIVO) { // el mozo esta disponible
            i = 0;

            while (i < mesas.size() && mesa.getNroMesa()!=mesas.get(i).getNroMesa())
                i++;

            if (i < mesas.size() && mesa.getNroMesa()==mesas.get(i).getNroMesa()) { // la mesa existe

                if (mesa.getEstado().equalsIgnoreCase("LIBRE") && !mesa.isAsignado()) { // el mozo y la mesa estan disponibles
                    mozo.addMesa(mesa);
                    mesa.setAsignado(true);

                }
                else { //la mesa no esta disponible
                    String msg;
                    if (mesa.isAsignado())
                        msg = "ERROR: La mesa ya estaba asignada";
                    else
                        msg="La mesa " + mesa.getNroMesa() + " no esta disponible";
                    throw new MesaNoDisponibleException(msg);
                }

            } else
                throw new MesaInexistenteException("La mesa seleccionada no existe");
        } else
            throw new MozoNoDisponibleException("El mozo " + mozo.getNya() + " no esta disponible");
    } else
        throw new MozoInexistenteException("El mozo seleccionado no existe");

    this.invariante();
}
```

En el repositorio de github en el paquete

“testCajaBlanca”

se encuentran todos los caminos realizados mediante una clase de JUnit, caminos con los que además de eClemma pudimos realizar y mostrar esta cobertura.

A modo de ejemplo de los caminos dejamos la siguiente imagen.

```
@Test
public void camino5() {

        Cerveceria.setInstance();
        Cerveceria cerveceria = Cerveceria.getInstance();
        Mozo mozo = new Mozo("Cristiano ronaldo", fecha1, 3);
        Mesa mesa = new Mesa(6, 1);

        try {
            cerveceria.addMozo(mozo);
            cerveceria.addMesa(mesa);
            cerveceria.asignarMesa(mozo, mesa);
            //OK

        } catch (MozoNoDisponibleException e) {
            Assert.fail();
            e.printStackTrace();
        } catch (MozoInexistenteException e) {
            Assert.fail();
            e.printStackTrace();
        } catch (MesaNoDisponibleException e) {
            Assert.fail();
            e.printStackTrace();
        } catch (MesaInexistenteException e) {
            Assert.fail();
            e.printStackTrace();
        } catch (MozoRepetidoException e) {
            Assert.fail();
            e.printStackTrace();
        } catch (MesaRepetidaException e) {
            Assert.fail();
            e.printStackTrace();
        }

    }
```

4. Test de Persistencia

Para el test de persistencia se tuvo en cuenta que la misma se realizó por serialización binaria, en un archivo llamado "`Cerveceria.bin`". Los métodos de testing utilizados de la clase PersistenciaTest dentro del paquete testPersistencia, abarcando distintos escenarios posteriormente mencionados, son los siguientes:

- **`testCrearArchivo()`:** Se testea la existencia del archivo de persistencia creado luego de llamar al método que persiste la cervecería.
- **`testDespersistirSinArchivo()`:** Se testea que la carga de la cervecería lance la excepción correspondiente si el archivo de persistencia no existe.
- **`testCerveceriaVacia()`:** Se testea la correcta persistencia y la posterior carga de la cervecería a partir del archivo persistido de la cervecería cuando tiene todas sus colecciones vacías.
- **`testCerveceriaNoVacia()`:** Se teste la correcta persistencia y posterior carga de la cervecería a partir del archivo persistido de la cervecería cuando contiene al menos un elemento en cada una de sus colecciones. Para este caso, se partió de la necesidad de no modificar el código del otro subgrupo, que no tenía redefinido el método `equals()` para sus clases. Por lo tanto, se procedió a comprobar la igualdad entre cada campo de los elementos de las colecciones antes de la persistencia y después de la despersistencia. Ésto se realizó para las colecciones mesas, mozos, productos y operarios.

En ninguno de los resultados de los tests mencionados se encontraron incongruencias con lo esperado, por lo que se pudo concluir que el test de persistencia arrojó que el trabajo de serialización fue realizado satisfactoriamente.

5. Test de GUI

En esta instancia, buscamos verificar que las funciones de la interfaz gráfica sean correctas y funcionen de manera adecuada.

Para realizar el test de GUI se utilizó la clase *Robot* de java que tiene como propósito la automatización de pruebas y actividades que involucren el mouse o el teclado sin la presencia o interacción física del usuario.

Con la clase robot pudimos automatizar muchas de las tareas involucradas en esta fase de testeo, ya que una prueba exhaustiva es imposible de llevar a cabo, este hecho, en el caso de las pruebas de GUI es aún más pronunciado, ya que la cantidad de secuencias de comandos que podría utilizar un cliente de la interfaz gráfica, es gigantesca.

Una de las dificultades que encontramos es que muchas validaciones en la interfaz, son realizadas mediante paneles de *JOptionPane* por ende se optó por implementar la siguiente solución para obtener los mensajes (y así poder testear su correcta salida).

1) Se hizo uso de la interfaz "*InterfazOptionPanel"*

2) La clase *"MiOptionPane"* como la clase *"FalsoOptionPane"* implementan esta interfaz.

3) El controlador posee un atributo de tipo *InterfazOptionPanel*.

4) Cada clase de testeo posee un atributo de tipo *FalsoOptionPane* el cual se le asigna al controlador.

Con los pasos realizados (los cuales fueron indicados por la cátedra) pudimos obtener cada uno de los mensajes de notificación de *JOptionPane* y así poder realizar un testeo más abarcativo.

En cuanto a las ventanas testeadas, como son muchas, concluimos en analizar sólo una de cada sección importante del programa, siendo éstas la ventana del login, la ventana de operaciones del operario, y como el administrador tiene como mayor característica la gestión (alta, baja y modificación), también testeamos la ventana de ABM de los productos.

Tanto para la interfaz del login como la de los productos se hizo un test de los botones y cuando están "Enabled" o "Disabled", en base a qué paneles de texto fueron rellenados (obteniendo una gran combinación de testeos para lo mencionado). Además se hizo un testeo con un conjunto de datos para estas ventanas, en la cual en base a un escenario con datos cargados, analizamos si las opciones de logueo o de registrar un producto se llevan a cabo exitosamente.

Para la interfaz de la ventana del operario nos encontramos con una *dificultad mucho mayor*, ya que para testear si los botones están o no activos, primero debemos seleccionar algún elemento de las listas de mozos o/y de mesas. Por ende tuvimos que recurrir a calcular las posiciones relativas de un elemento de las lista en su panel, y luego *programar al "Robot" para que simule el movimiento del mouse para ir hasta esa coordenada, y luego presionar dicho elemento.*

Una vez sobrepasada esta dificultad pudimos testear tanto que los botones se habiliten o no al seleccionar un elemento de la lista, como a que las asignaciones entre dos elementos de listas seleccionadas sean correctos.

En total se realizaron *28 tests de GUI*, otorgando todos un resultado correcto.

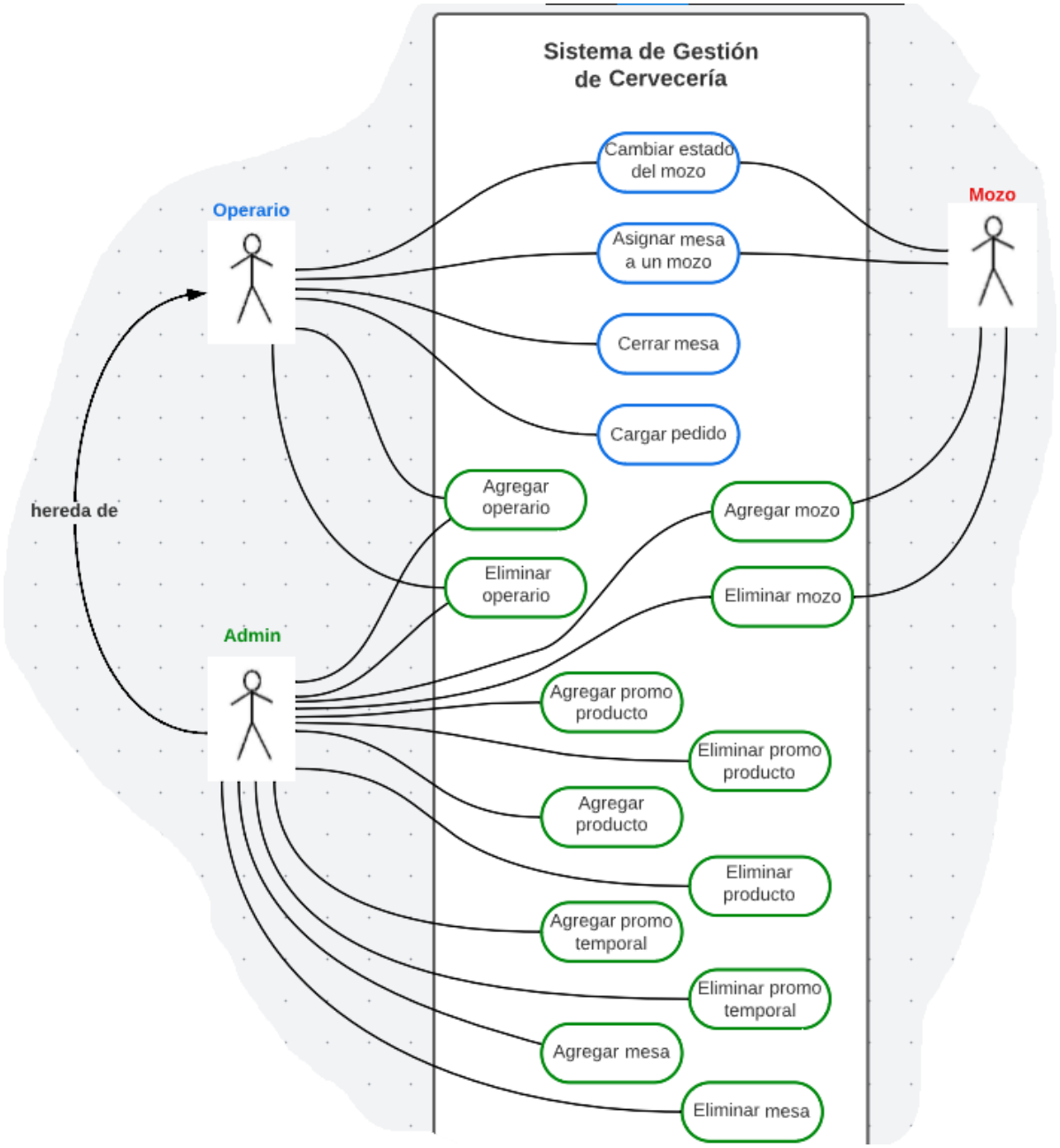
En el video muestran con detalles estos tests realizados, pero además se pueden encontrar en el paquete

"testGUI".

del repositorio adjunto de github.

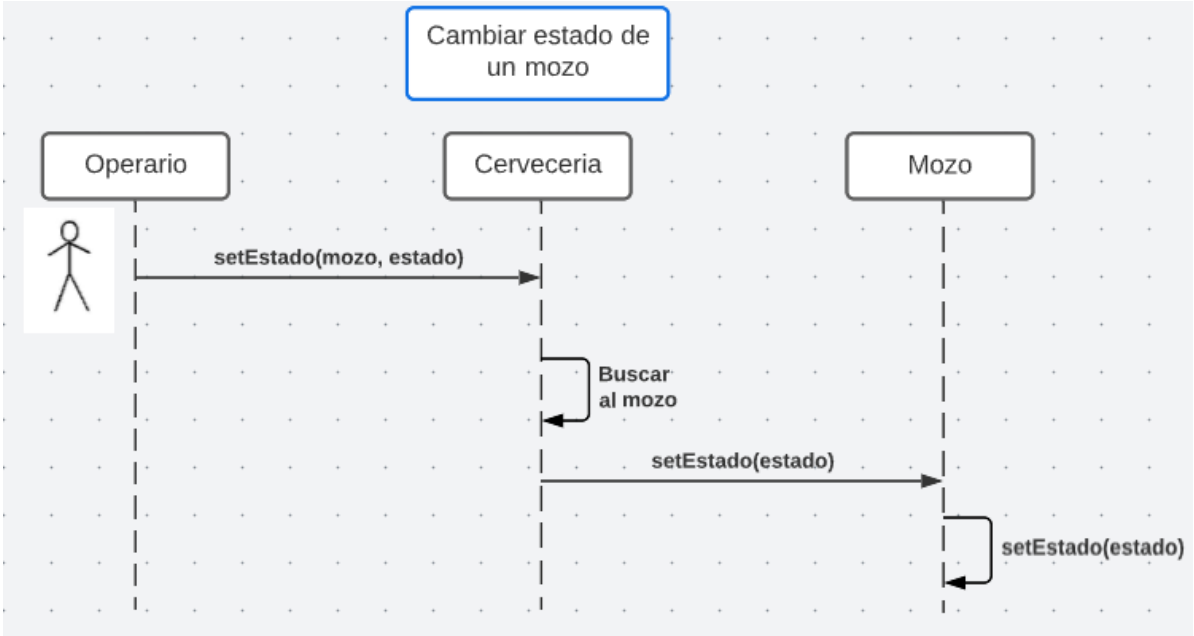
6. Test de integración

Para las pruebas de integración, se decidió optar por el análisis orientado a objetos, para poner el foco en la interacción entre los módulos que ya fueron testeados mediante las pruebas unitarias de caja negra y caja blanca. Los casos de uso planteados se basan en cómo un operario o un administrador interactúa con el sistema de la cervecería para llevar a cabo las distintas operaciones. Éstos pueden observarse en el siguiente diagrama:



Para ilustrar un proceso normal dentro del sistema, se muestran a continuación los diagramas de secuencia de los casos de uso utilizados en un escenario en el cual el local abre sus puertas, asigna su estado y una mesa a un mozo, llega una familia a comer abriendo la comanda correspondiente y finalmente cerrando la mesa. Junto a los diagramas, se pondrá la información acerca de los casos de prueba utilizados.

Caso de uso

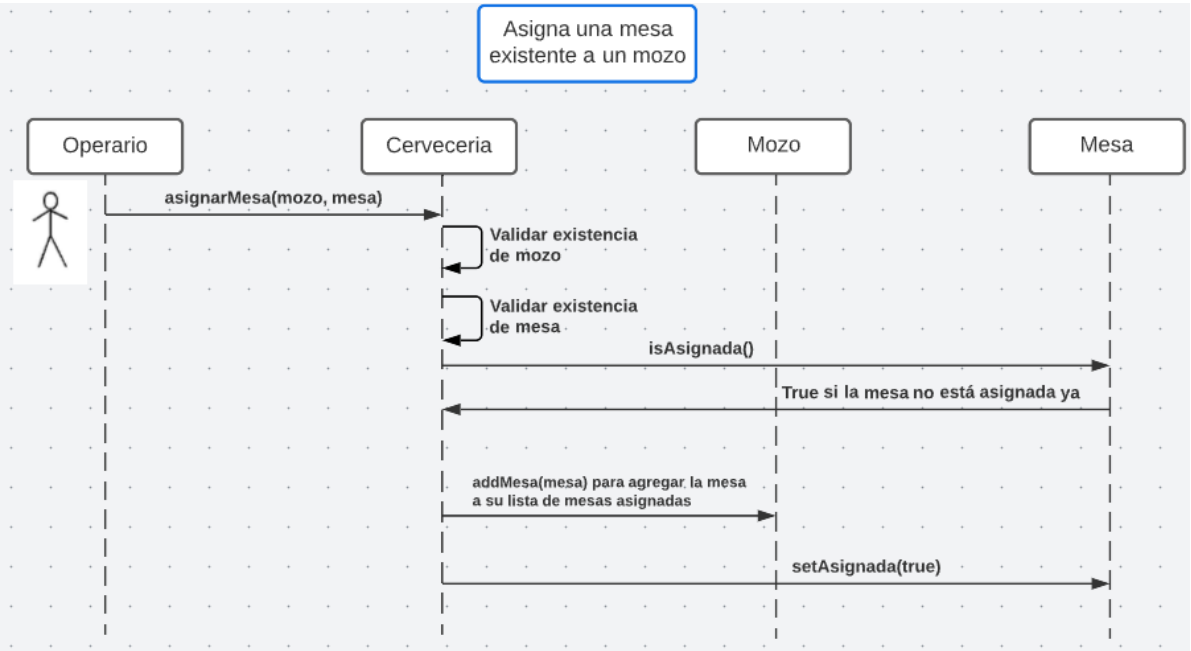


Caso de prueba

Escenario	Descripción
1	Lista de mozos vacía.
2	Un mozo ausente.

Caso	Entrada	Salida esperada	Salida obtenida	Escenario
1	mozo = cualquiera estado = activo	MozoInexistenteException()	Sin errores	1
2	mozo = el único mozo estado = activo	Se cambió su estado a activo.	Sin errores	2

Caso de uso

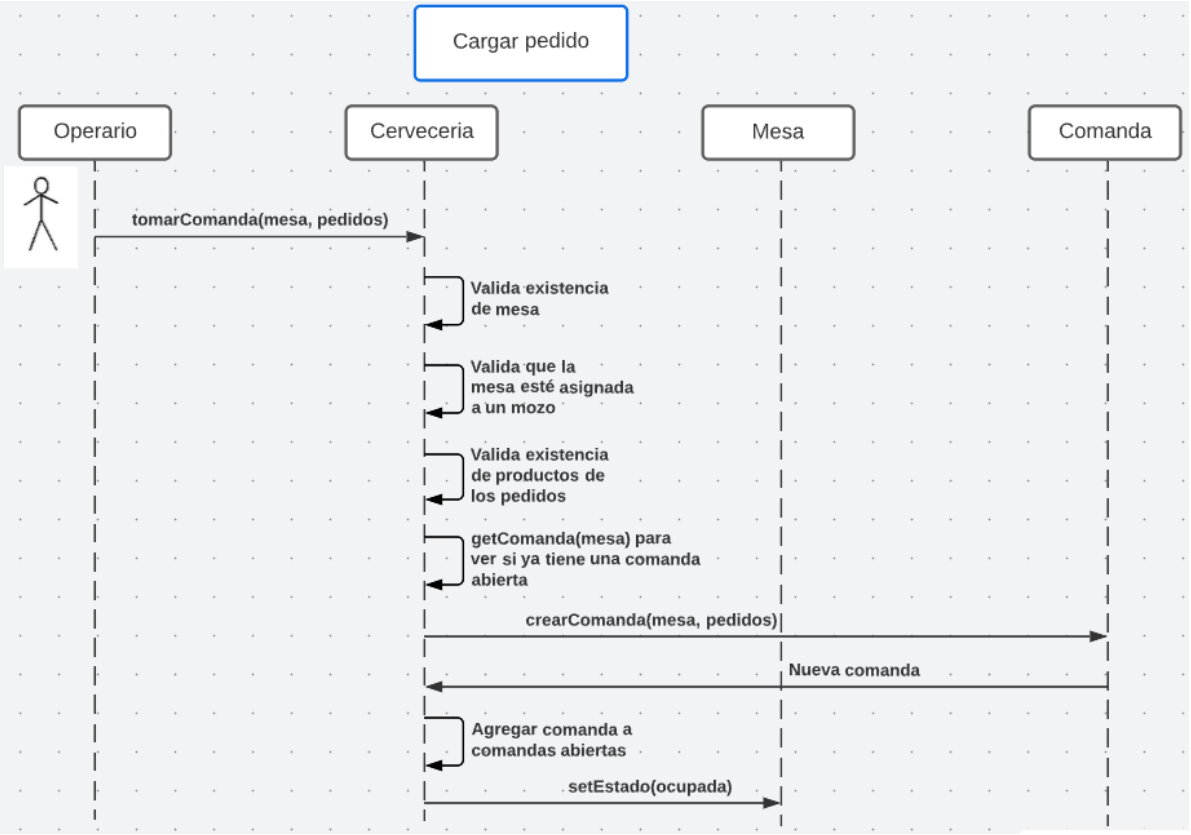


Caso de prueba

Escenario	Descripción
1	Lista de mozos y mesas vacías.
2	Un mozo de franco y una mesa.
3	Un mozo activo y una mesa (barra) sin asignar.
4	2 mozos activos y una mesa asignada al mozo 1.

Caso	Entrada	Salida esperada	Salida obtenida	Escenario
1	mozo = cualquiera mesa = cualquiera	MozoInexistenteException()	Sin errores	1
2	mozo = único mozo mesa = única mesa	MozoNoDisponibleException()	Sin errores	2
3	mozo = único mozo mesa = mesa 1	MesaInexistenteException()	Sin errores	3
4	mozo = mozo 2 mesa = única mesa	MesaNoDisponibleException()	Sin errores	4
5	mozo = único mozo mesa = única mesa	Se asigna correctamente.	Sin errores	3

Caso de uso

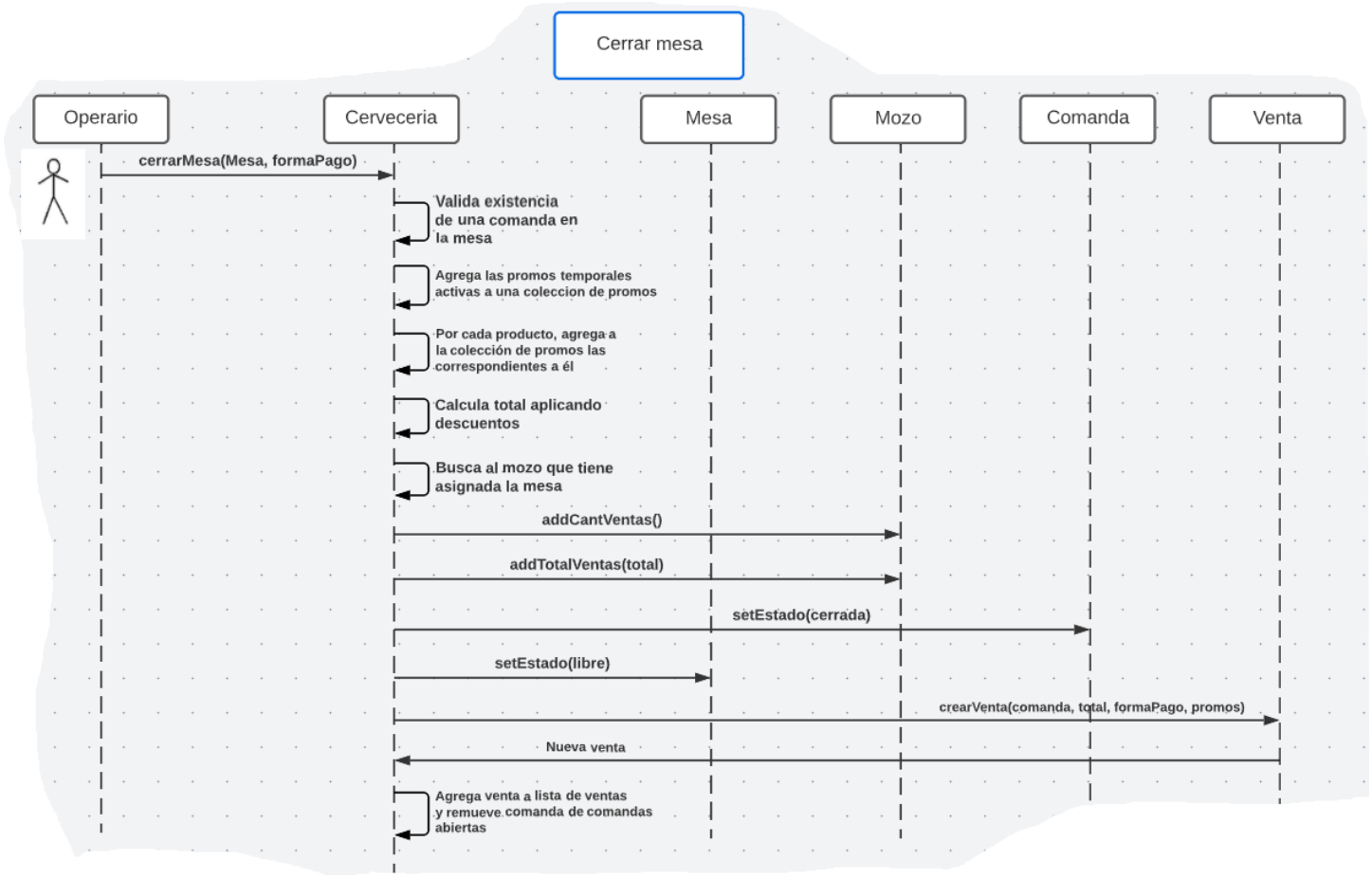


Caso de prueba

Escenario	Descripción
1	Mesa 1 asignada y libre, Mesa 2 asignada y ocupada, Mesa 3 no asignada.

Caso	Entrada	Salida esperada	Salida obtenida	Escenario
1	mesa = mesa 3 Lista de pedidos válidos	MesaNoDisponibleException()	Sin errores	1
2	mesa = mesa 4 Lista de pedidos válidos	MesaInexistenteException()	Sin errores	1
3	mesa = mesa 1 Lista de pedidos inválidos	ProductosInvalidosException()	Sin errores	1
4	mesa = mesa 1 Lista de pedidos	Setea mesa en ocupada, crea una comanda con una lista de productos válidos y la añade a la lista de comandas abiertas.	Sin errores	1
5	mesa = mesa 2 Lista de pedidos	Actualiza la lista de pedidos de la comanda de la mesa.	Sin errores	1

Caso de uso



Caso de prueba

Escenario	Descripción
1	Mesa 1 asignada y libre, Mesa 2 asignada y ocupada.

Caso	Entrada	Salida esperada	Salida obtenida	Escenario
1	mesa = mesa 1 formaPago=”efectivo”	MesaSinComandaException()	Sin errores	1
2	mesa = mesa 2 formaPago=”nada”	Se esperaría una excepción de forma de pago inexistente.	El programa no valida la forma de pago.	1
3	mesa = mesa 2 formaPago=”efectivo”	Actualiza las ventas del mozo encargado de la mesa, cierra la comanda creando una venta y agregándola a la lista de ventas y setea la mesa en libre.	Se calcula mal el descuento de las promos temporales. Si la promo es del 30% nos devuelve el total con un 70% de descuento.	1

7. Conclusión

Durante el transcurso del testing, se tuvieron en cuenta las distintas metodologías vistas en la materia. Sin quitar mérito sobre el trabajo testeado, cada técnica aportó información congruente, dando a luz algunos errores de implementación que, de no ser por estas herramientas, no se hubieran detectado.

Se realizó el test de Caja Negra sobre todos los métodos significativos del proyecto, utilizando su documentación para encontrar casos de prueba correspondientes a estos. Luego, se determinaron clases válidas e inválidas para varios de los métodos testeados. Gracias a las pruebas de unidad se pudieron determinar errores generales, para luego poder profundizar los testeos y descubrir el origen de aquellos. Es un método útil para reconocer grandes fallos, pero que se ve afectado cuando la documentación del programa es escasa, ya que dificulta la determinación de entradas, salidas, y posibles errores en la ejecución.

Se aplicó el test de Caja Blanca sobre un método que poseía poca cobertura y es muy notable en el flujo del programa. Se determinó su complejidad ciclomática en base a su Grafo ciclomatico. Una vez realizados estos pudimos observar una mejoría en cuanto a la cobertura del código.

En el test de GUI se probaron las distintas alternativas que podían surgir a partir de la interfaz dada. Se verificó que los componentes cumplieran su función principal.

El test de persistencia requirió de realizar algunas modificaciones menores a la capa de modelo para que este funcionara correctamente, pero una vez aplicados los cambios mencionados previamente, los test resultaron exitosos.

Por último, en el test de Integración, se utilizaron pruebas basadas en el análisis orientado a objetos, dando así lugar a casos en los que se integran múltiples módulos del sistema para verificar las funcionalidades en conjunto.

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, pues, de otra manera, las tareas de testeo pueden dar lugar a resultados difusos.

Para concluir se puede afirmar que se aplicaron los conceptos brindados a lo largo del cuatrimestre por la cátedra en los *aproximadamente 80 tests* realizados, y que durante el transcurso de este trabajo práctico muchos de estos conceptos fueron afianzados y estudiados con mayor detenimiento para así poder cumplir el objetivo del proyecto.