TALLER DE PROGRAMACIÓN I

Grupo 1 - Subgrupo 1

Realizaron el código:

- Ignacio Suarez Quilis
- Tomas Matarazzo

Realizaron el testing:

- Lautaro Nahuel Bruses
- Matías Angélico

Video presentación:

https://youtu.be/DBXOpo-g_yI

Repositorio testeado:

https://github.com/TomasMatarazzo/Cerveceria10

1. Índice

2. Introducción

Se define al **testing** como un proceso para verificar y validar la funcionalidad de un programa o una aplicación de software con el objetivo de garantizar que el producto contenga la menor cantidad de defectos posibles. Consiste en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados a partir del dominio de ejecución que usualmente es infinito, en relación con el comportamiento esperado.

El Testing no es una actividad que se piensa al final del desarrollo del software, sino que va paralelo a este. El programa se prueba ejecutando solo unos pocos casos de prueba dado que por lo general es física, económica y técnicamente imposible ejecutarlo para todos los valores de entrada posibles.

En este informe, se expondrán diferentes pruebas funcionales como “Pruebas unitarias de Caja Negra”, “Pruebas unitarias de Caja Blanca”, “Test de Persistencia”, “Test de Interfaces Graficas de Usuario” y “Test de Integración”, aplicadas a un sistema de gestión de una cervecería.

3. Desarrollo

3.1. Pruebas unitarias de Caja Negra

Las pruebas de caja negra, también llamadas de comportamiento o Funcionales se basan en lo que se espera de un módulo, fundamentándose en la especificación de requerimientos.

Para la realización de estas pruebas y poder realizarlas de forma automática se utilizó la librería *“JUnit5”* la cual permitió desarrollar conjuntos de pruebas de forma sencilla y sistemática.

En base a la documentación del código presentada se decidió probar los métodos de la clase Empresa, ya que es la más representativa del proyecto y la que realiza la mayoría de las funciones del sistema.

Cabe destacar, que no se incluyó el testeo de los métodos getters, setters, toString, equals y hashcode, por no aportar datos relevantes respecto del método, ya que en su mayoría, fueron generados por el entorno de desarrollo.

A modo de ejemplo se seleccionó un método para enseñar el proceso para la realización de la prueba.

- ★ **Método a Testear:** Agregar Mozo

- ★ **Clase:** Empresa
- ★ **Parámetros:**
  - ■ **Nombre:** String
  - ■ **FechaNacimiento:** Date
  - ■ **CantidadHijos:** entero
  - ■ **Estado:** entero.

Precondiciones:

- Nombre - distinto de vacío o null
- FechaNacimiento - distinto de vacío o null
- CantidadHijos - distinto de vacío o null
- Estado - distinto de vacío o null
- La colección de mozos debe existir

Se planteó los escenarios posibles para este método :

ESCENARIOS	
Nro de escenario	**Descripción**
Escenario 1	Conjunto de Mozos vacío
Escenario 2	Conjunto de Mozos con elementos

Luego la Tabla de Particiones para cada Escenario:

TABLA DE PARTICIONES		
Condición de entrada	**Clases válidas**	**Clases inválidas**
Nombre	Nombre VALIDO 1	solo clases válidas
Fecha de Nacimiento	Fecha de Nacimiento VÁLIDA > 18 años de edad 3	FechaNacimiento NO VÁLIDA 4.1 FechaNacimiento < 18 años de edad 4.2
Cantidad de hijos	Cantidad de Hijos >= 0 5	cant hijos <0 6
Estado	Estado VALIDO 7	Estado INVÁLIDO 8

Y por último la batería de pruebas con las cuales hacer los test unitarios

BATERÍA DE PRUEBAS			
Tipo de clase (correcta o incorrecta)	**Valores de entrada**	**Salida esperada**	**Clases de prueba cubiertas**
Correcta	“Tomas Matarazzo” , 2/2/1999 , 0, 1	Agrega mozo a colección	1, 3 , 5 , 7
Incorrecta	“Tomas Matarazzo” , 2/2/1999 , -1, 1	“ERROR → cantidad de hijos negativa”	6
Incorrecta	“Tomas Matarazzo” , 34/22/1999 , 0, 1	“ERROR → fecha incorrecta”	4.1
Incorrecta	“Tomas Matarazzo” , 2/2/2015 , 0, 1	“ERROR → menor de edad”	4.2
Incorrecta	“Tomas Matarazzo” , 2/2/1999 , 0, 4	“ERROR → estado invalido”	8

Para concluir, las pruebas unitarias realizadas sólo arrojaron falla en el método que se encarga de agregar productos, ya que permite crear objetos Producto con stock negativo, sin lanzar excepción.

El resto de los módulos presentaron leves fallas al testearlos, pero que fueron corregidas dentro del código de modo que superen las pruebas.

Con el objetivo de facilitar su visualización se generó el “***Test Report***” que contiene los resultados de todos las pruebas realizadas.

COBERTURA	
Cobertura de Clases	100 %
Cobertura de Métodos	61,4 %
Cobertura de Lineas	78,7 %

3.2. Pruebas unitarias de Caja Blanca

Las pruebas de Caja Blanca se encargan de analizar el diseño, el código y la estructura interna de cada unidad. Estas pruebas se basan en los detalles referentes al código fuente, con el objetivo de generar casos de prueba que abarquen la mayor cobertura posible de la unidad analizada.

Finalizadas las pruebas de Caja Negra, se generó el *"**Coverage Report**"* en donde se encontró un método, que había sido testeado, con líneas del código que no habían sido abarcadas por ningún caso de prueba utilizado. Por lo que se decidió realizar una prueba de caja blanca al método *setTotal()*, dentro de la clase Factura.

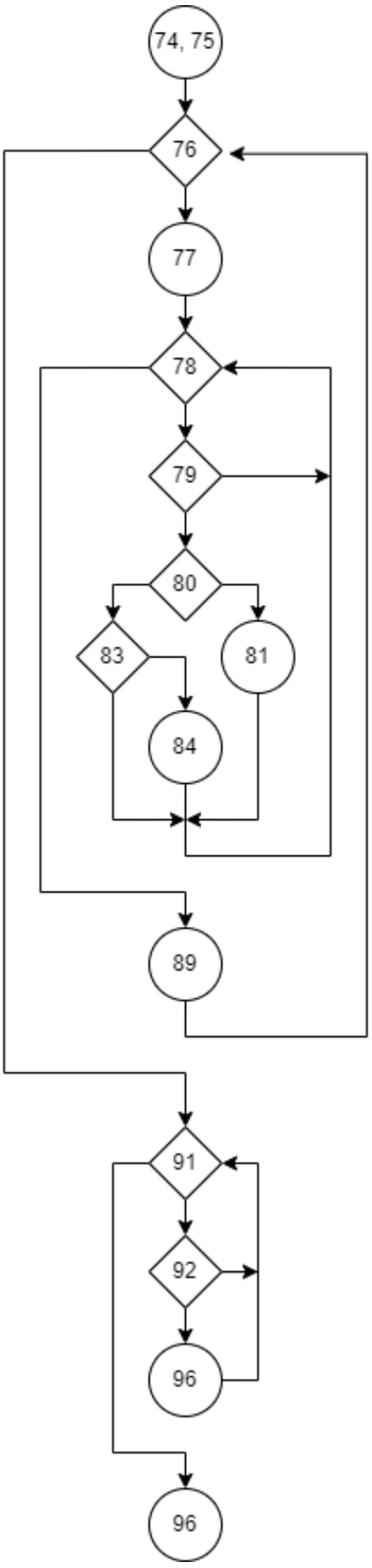
```
66    /**
67     * El metodo permite calcular el total a facturar, recorre todos los pedidos realizados y hace un calculo parcial al que posteriormente se le van
68     * a aplicar descuentos en caso de ser necesario. Primero se recorre el arreglo de promociones de producto para saber si es necesario
69     * aplicarle un descuento, para esto la promocion debera estar activa. Luego recorreremos el arreglo de promociones temporales para
70     * saber si corresponde aplicarselas.
71     * @return devuelve un flotante que debera ser mayor a cero, representa el total a cobrar a la mesa correspondiente
72     */
73    public double setTotal() {
74        double total=0;
75        double parcial;
76
77        for (int i=0;i<pedidos.size();i++) {
78           parcial=pedidos.get(i).getCantidad()*pedidos.get(i).getProducto().getPrecioVenta();
79            if (this.promocionesProductos != null) { // || promocionesProductos.size() > 0) {
80                for (int j = 0; j < promocionesProductos.size(); j++) {
81                    if (promocionesProductos.get(j).isActiva()) {
82                        if (promocionesProductos.get(j).isAplicaDosPorUno()) {
83                            parcial /= 2.;
84                        } else if (promocionesProductos.get(j).isAplicaDtoPorCantidad() && pedidos.get(i).getCantidad() >= promocionesProductos.get(j).getDtoPorCantidad_CantMinima()) {
85                            parcial = promocionesProductos.get(j).getDtoPorCantidad_PrecioUnitario() * pedidos.get(i).getCantidad();
86                        }
87                    }
88                }
89            }
90            total += parcial;
91        }
92
93        if (this.promocionesTemporales != null){ //|| promocionesTemporales.size() > 0) {
94            for (int k = 0; k < promocionesTemporales.size(); k++) {
95                if (promocionesTemporales.get(k).isActivo() && promocionesTemporales.get(k).getFormaDePago().equals(this.getFormaDePago()) && promocionesTemporales.get(k).getDiasDePromo() == Date.from(Instant.now()).getDay()) {
96                    total = total - total*(double) promocionesTemporales.get(k).getPorcentajeDescuento() / 100;
97                }
98            }
99        }
100
101       return total;
102   }
```

Cobertura de código luego de aplicar la prueba de Caja Negra

Se comenzó por analizar la funcionalidad del método a partir de su documentación. Rápidamente se pudo notar que el código tenía bastantes condiciones redundantes, por lo que se procedió a removerlas y a simplificar la complejidad del algoritmo, obteniendo la siguiente cobertura:

```
66    /**
67     * El metodo permite calcular el total a facturar, recorre todos los pedidos realizados y hace un calculo parcial al que posteriormente se le van
68     * a aplicar descuentos en caso de ser necesario. Primero se recorre el arreglo de promociones de producto para saber si es necesario
69     * aplicarle un descuento, para esto la promocion debera estar activa. Luego recorreremos el arreglo de promociones temporales para
70     * saber si corresponde aplicarselas.
71     * @return devuelve un flotante que debera ser mayor a cero, representa el total a cobrar a la mesa correspondiente
72     */
73    public double setTotal() {
74        double total=0;
75        double parcial;
76
77        for (int i=0;i<pedidos.size();i++) {
78           parcial=pedidos.get(i).getCantidad()*pedidos.get(i).getProducto().getPrecioVenta();
79            if (this.promocionesProductos != null) { // || promocionesProductos.size() > 0) {
80                for (int j = 0; j < promocionesProductos.size(); j++) {
81                    if (promocionesProductos.get(j).isActiva()) {
82                        if (promocionesProductos.get(j).isAplicaDosPorUno()) {
83                            parcial /= 2.;
84                        } else if (promocionesProductos.get(j).isAplicaDtoPorCantidad() && pedidos.get(i).getCantidad() >= promocionesProductos.get(j).getDtoPorCantidad_CantMinima()) {
85                            parcial = promocionesProductos.get(j).getDtoPorCantidad_PrecioUnitario() * pedidos.get(i).getCantidad();
86                        }
87                    }
88                }
89            }
90            total += parcial;
91        }
92
93        if (this.promocionesTemporales != null){ //|| promocionesTemporales.size() > 0) {
94            for (int k = 0; k < promocionesTemporales.size(); k++) {
95                if (promocionesTemporales.get(k).isActivo() && promocionesTemporales.get(k).getFormaDePago().equals(this.getFormaDePago()) && promocionesTemporales.get(k).getDiasDePromo() == Date.from(Instant.now())
96                    total = total - total*(double) promocionesTemporales.get(k).getPorcentajeDescuento() / 100;
97                }
98            }
99        }
100
101       return total;
102   }
```

Cobertura de código luego de simplificar su complejidad ciclomática



Grafo de control del método setTotal()

Posteriormente se diseñó un grafo de control para representar el flujo de ejecución del método para así poder determinar la cobertura del código. Para la construcción del grafo se utilizó el *"Método descendente"*, comenzando por el principio del algoritmo, aprovechando la secuencialidad de la ejecución del mismo.

Se observó que al tener 7 nodos de decisión, la complejidad ciclomática del método será de 8.

A continuación se definió el conjunto básico de caminos independientes a partir del *"Método simplificado"*.

1. **C1: (74, 75) - (76) - (91) - (96)**
2. **C2:** (74, 75) - (76) - **(77) - (78) - (89)** - (76) - (91) - (96)
3. **C3:** (74, 75) - (76) - (77) - (78) - **(79)** - (78) - (89) - (76) - (91) - (96)
4. **C4:** (74, 75) - (76) - (77) - (78) - (79) - **(80) - (81)** - (78) - (89) - (76) - (91) - (96)
5. **C5:** (74, 75) - (76) - (77) - (78) - (79) - (80) - **(83)** - (78) - (89) - (76) - (91) - (96)
6. **C6:** (74, 75) - (76) - (77) - (78) - (79) - (80) - (83) - **(84)** - (78) - (89) - (76) - (91) - (96)
7. **C7:** (74, 75) - (76) - (91) - **(92)** - (91) - (96)
8. **C8:** (74, 75) - (76) - (91) - (92) - **(93)** - (91) - (96)

Utilizando este método obtenemos la totalidad de los caminos independientes del algoritmo, pero si lo que buscamos es obtener el 100% de cobertura solo necesitaremos 3 caminos:

1. **C4:** (74, 75) - (76) - (77) - (78) - (79) - (80) - (81) - (78) - (89) - (76) - (91) - (96)
2. **C6:** (74, 75) - (76) - (77) - (78) - (79) - (80) - (83) - (84) - (78) - (89) - (76) - (91) - (96)
3. **C8:** (74, 75) - (76) - (91) - (92) - (93) - (91) - (96)

Con este resultado se determinaron los casos de prueba con los valores de entrada correspondientes a cada camino, planteando los siguientes escenarios:

- **Escenario 1**: Arraylist de promocionesTemporales con una promoción temporal activa, con FormaDePago = "Efectivo", diasDePromo = {"Lunes"} y un porcentajeDescuento = 50. Arraylist de promocionesProductos vacía.
- **Escenario 2:** Arraylist de promocionesTemporales vacía. Arraylist de promocionesProductos con 2 promociones de producto activa. Una con aplicaDosPorUno = true y producto = "Coca-Cola" y la otra con aplicaDtoPorCantidad = true, producto = "Pepsi" , una CantMinima = 2 y un PrecioUnitario = 200. Ambas promociones con diasDePromo = "Lunes".

Resultando en la siguiente tabla de particiones:

Camino	Escenario	Parámetros de entrada	Salida esperada
C4	2	pedidos = {p1 = {"Coca-Cola", 100, 150, 100}, 2}, FormaDePago = "Efectivo",	total = 150

C6	2	pedidos = {p2 = {“Pepsi”, 200, 250, 100}, 2}, FormaDePago = “Efectivo”,	total = 400
C8	1	pedidos = {p1 = {“Coca-Cola”, 100, 150, 100}, 2}, FormaDePago = “Efectivo”	total = 150

Aclaración: si bien el método no recibe parámetros en si, entendemos que el ArrayList de pedidos representa un parámetro del método, ya que es un atributo exclusivamente de la Factura. Lo mismo ocurre con el atributo FormaDePago.

A raíz de esta prueba determinamos varias incongruencias en el código, tales como:

★ Se aplican promociones temporales aun sin pedidos en la factura (tampoco se aclara como precondición del método).
★ El tipo de atributo “diasDePromo” no coincide en ambas promociones, en uno es un atributo int en otro un ArrayList de String.
★ Al momento de aplicar el descuento “aplicaDosPorUno” nunca verifica que el atributo “producto” de la factura y la promocionProducto coincidan.

Se llegó a la conclusión de no seguir avanzando con la prueba, ya que el método no realiza correctamente la funcionalidad pedida, además de que contradice en varias oportunidades al contrato y carece de precondiciones claras. Lo mejor es volver a formular el problema del dominio desde cero y codificar el algoritmo de nuevo.

3.3. Test de Persistencia

La persistencia permite recuperar información desde un sistema de almacenamiento no volátil y hacer que esta persista. La serialización nos permite guardar el estado de un componente en disco, abandonar el Entorno Integrado de Desarrollo (IDE) y restaurar el estado de dicho componente cuando se vuelve a correr el IDE.

El sistema implementa una persistencia de tipo binaria, la cual se realiza sobre la totalidad de la cervecería, ya que dentro de esta se almacenan todas las instancias e información necesaria para el buen funcionamiento del sistema.

Esta propiedad se testeó en la clase *“PersistensiaTest”*, donde se plantearon los siguientes 2 escenarios:

- El archivo binario *“Cerveceria.bin”* existe
- El archivo binario *“Cerveceria.bin”* NO existe

Para el primer escenario se plantearon 2 casos de prueba, donde se probó leer y escribir, primero una empresa vacía y luego una empresa con

información contenida dentro. Para esto, se sobreescribieron los métodos *equals()* y *hashcode()* en la clase empresa para poder comparar dos instancias de la clase empresa.

Luego, se plantearon 4 casos de prueba para el escenario restante, en donde, en primer lugar, se probó crear el archivo desde cero, obteniendo éxito. Se comprobó que leer o escribir un archivo inexistente, lanza su debida excepción. Y también que al intentar abrir el archivo inexistente, los métodos devuelven una excepción.

3.4. Test de Interfaces Graficas de Usuario

Para poder probar las interfaces gráficas se propuso separar las reglas del negocio, de forma que se pueda testear por separado, sin ser afectado por la GUI. Además se procuró la utilización de componentes estándar, que se puedan considerar ya testeados.

Se utilizó la clase robot, la cual permitió simular eventos de teclado y mouse, sumado a la creación de la clase UtilsTest con la cual, mediante métodos estáticos, se pudo realizar las tareas más comunes, como por ejemplo hacer click en un botón, ingresar un texto, como así también obtener la referencia de componentes mediante un nombre previamente seteado.

Los test de GUI se hicieron sobre la ventana Principal, la cual tiene según el escenario (si quien accede es un Operario o un Administrador), 7 u 8 botones.

Ventana VistaPrincipal

Mediante el uso del robot se verificó que los botones correspondientes estuviesen habilitados o deshabilitados al abrir la ventana.

Luego se probó que al iniciar la jornada (estimulación del botón), aquellos botones que estaban deshabilitados se habilitarán y además que aquellos que estaban habilitados no cambien su estado.

Continuamos la prueba comprobando que cada uno de los botones al ser estimulado lance la ventana correcta.

Todos los test realizados, en cada uno de los escenarios, fueron correctos y no arrojaron fallas.

3.5. Test de Integración

Las pruebas de integración se basan en diseñar la interacción entre los distintos componentes de un sistema, con el objetivo de hallar errores de programación entre componentes y/o errores de interoperabilidad.

Para realizar esta prueba se optó por tomar un enfoque *"Orientado a Objetos"*, que se enfoca en la interacción entre unidades, suponiendo que cada una fue aprobada a nivel de unidad con anterioridad.

3.5.1. Casos de Uso

En primer lugar, se definirán los casos de uso que utilizaremos en la prueba, optando por los que abarquen más unidades del sistema, describiendo las interacciones que conducen a lograr o abandonar el objetivo, y representen mejor una funcionalidad pura y exclusiva de este sistema.

CU1 - Crear Comanda

Descripción: El operario desea asociar una nueva comanda con pedidos a una mesa libre con un mozo activo asociado.
Actores Principales: Operario
Precondiciones:

I. El operario debe estar logueado en el sistema.
II. La cervecería tiene que tener mesas habilitadas.
III. La cervecería tiene que tener mozos activos.
IV. La cervecería tiene que tener productos en venta con stock disponible.
V. La cervecería tiene que tener al menos 2 productos con una promoción activa.
VI. La mesa asociada debe estar libre.
VII. La mesa asociada debe tener un mozo activo asociado.

Flujo Normal:

1. El operario accede a la ventana 'nueva comanda'.
2. El sistema solicita una mesa, un mozo y un conjunto de pedidos.
3. El operario ingresa una mesa, un mozo y los pedidos deseados.
4. El sistema valida que el estado de la mesa y el mozo ingresados sea el correcto.
5. El sistema almacena la nueva comanda y cierra la ventana.

Flujo Alternativo 1:

1. El sistema no cuenta con pedidos creados con anterioridad.
2. El operario crea los pedidos que desee asignar a la comanda.

Excepciones:

- **E1**: El estado de la mesa ingresada es 'Ocupada' o distinto de 'Libre'. El sistema notifica el error.

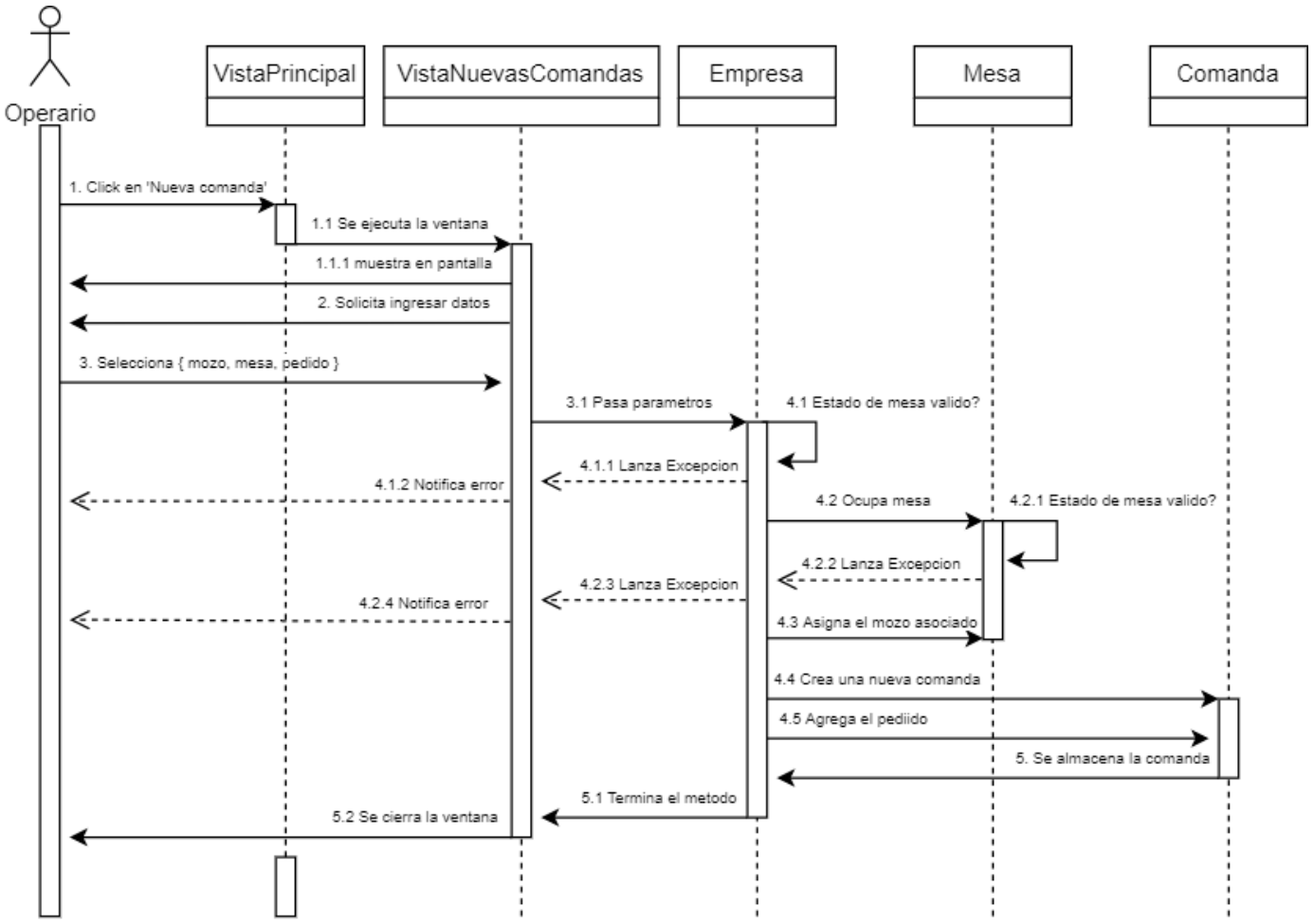
Postcondiciones:

I. Una nueva comanda activa es agregada al sistema.
II. La mesa asociada pasa a estar ocupada.
III. El stock de los productos agregados a los pedidos es actualizado.

3.5.2. Diagrama de Secuencia

Se utilizó un diagrama de secuencia para representar los distintos flujos del caso de uso, así también como la interacción de los objetos que intervienen, a través del tiempo. En él, se planteó con detalle, la implementación del escenario, incluyendo los objetos y las clases que se usan para su implementación y los mensajes que se intercambian entre los objetos del dominio.

En el siguiente diagrama se modeló el flujo normal de ejecución del método *altaComanda()* presente en el código:



3.5.3. Casos de Prueba

A partir del caso de uso planteado y del diagrama de secuencia realizado, se pueden deducir varios casos de prueba. Para ello se determinaron:

1) El estado de los diversos componentes

Tipo de elemento	Posibles estados
VistaPrincipal	Presente, ausente
VistaNuevasComandas	Presente, ausente
Empresa	Presente, ausente
Mesa	Presente, ausente

2) Los posibles valores que pueden tomar los diversos parámetros de los métodos invocados.

A partir de la vista se reciben los datos y el controlador se encarga de pasar solo 3 parámetros al método *'altaComanda()'*, por lo que sólo se considerarán estos 3:

Parámetros	Valores determinantes
Mesa	“Libre” o “Ocupada”
Mozo	“Activo” o “Franco” o “Ausente”
Pedido	{existe}

3) Conjunto de datos adecuado:

Entrada	Condiciones de entrada	Salida esperada
Mesa = {...,”Libre”}, Mozo = {...,“Activo”}, pedido = {Producto,... }	VistaPrincipal ausente	Falla de ejecución
Mesa = {...,”Libre”}, Mozo = {...,“Activo”}, pedido = {Producto,... }	VistaNuevasComandas ausente	Falla de ejecución
Mesa = {...,”Libre”}, Mozo = {...,“Activo”}, pedido = {Producto,... }	Empresa ausente	Falla de ejecución
Mesa = {...,”Libre”},	Mesa ausente	Falla de ejecución

Mozo = {...,“Activo”}, pedido = {Producto,... }		
Mesa = {...,”Libre”}, Mozo = {...,“Activo”}, pedido = {Producto,... }	Comanda ausente	Falla de ejecución
Mesa = {...,”Libre”}, Mozo = {...,“Activo”}, pedido = {Producto,... }	VistaPrincipal visible VistaNuevasComandas visible Empresa presente Mesa presente	Nueva comanda creada
Mesa = {...,”Ocupada”}, Mozo = {...,“Activo”}, pedido = {Producto,... }	VistaPrincipal visible VistaNuevasComandas visible Empresa presente Mesa presente	Mensaje de error
Mesa = {...,”Libre”}, Mozo = {...,“Franco”}, pedido = {Producto,... }	VistaPrincipal visible VistaNuevasComandas visible Empresa presente Mesa presente	Mensaje de error
Mesa = {...,”Libre”}, Mozo = {...,“Ausente”}, pedido = {Producto,... }	VistaPrincipal visible VistaNuevasComandas visible Empresa presente Mesa presente	Mensaje de error

A partir de este proceso, se pudo notar que existen contradicciones entre el contrato establecido en los requerimientos y la documentación del método, la cual es excesivamente escasa.

Sin embargo, a raíz de esto, se realizaron los test pertinentes para abarcar todos los casos de prueba definidos. Exceptuando los test que ya se han realizado en las pruebas unitarias.

4. Conclusiones

A partir del trabajo realizado llegamos a la conclusión principal de que es indispensable contar con una documentación extensa, precisa y completa, para poder realizar un buen trabajo de testing. De lo contrario, las pruebas no resultarán del todo efectivas, llegando incluso al punto de no poder realizar pruebas a ciertos métodos, en donde la documentación es incoherente y poco fiel a los requerimientos del sistema.

Además es necesario tener una especificación de requerimiento de software bien detallada y que la misma no deje zonas grises, para que ambos documentos no se contradigan en ningún punto.

En función de este informe, se observa que el sistema testeado tiene muchos vacíos e incógnitas dentro de su documentación, que encabezan muchos métodos que son incoherentes a los requerimientos extendidos por la cátedra y no abarcan la totalidad de su funcionalidad. Incluso luego de modificar el código para que cumpla con ciertos requerimientos, el sistema continúa fallando.