# SISTEMA DE CERVECERÍA

## TALLER DE PROGRAMACIÓN I
## UNMDP


**SUBGRUPO**
-BOOLLS, NICOLÁS
-GUTIÉRREZ, ALAN

**INTEGRANTES**
-BOOLLS, NICOLÁS
-GUTIÉRREZ, ALAN
-RODRÍGUEZ, JUAN GABRIEL
-UZQUIANO, TOMÁS EMANUEL

# ÍNDICE

# Introducción

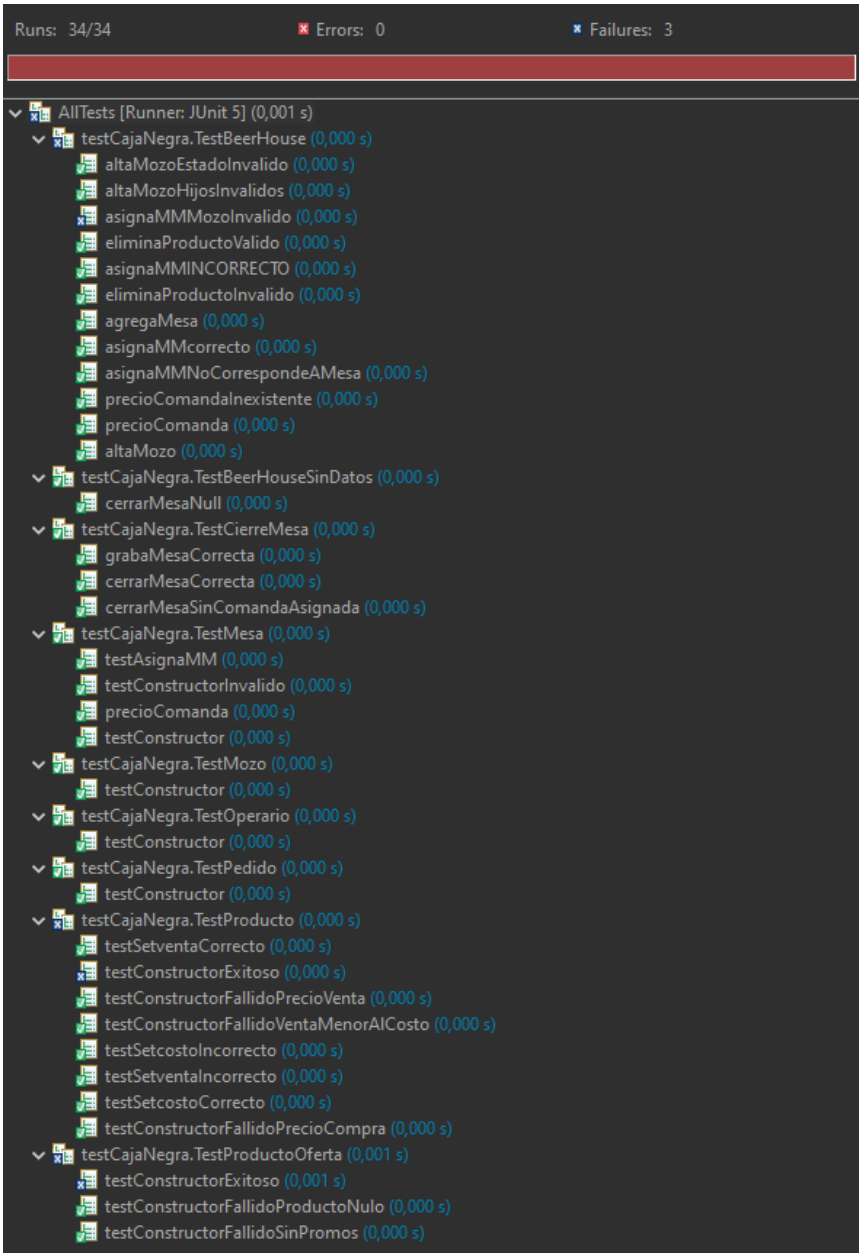
En el presente informe se detallarán los resultados obtenidos tras aplicar técnicas de testing en un sistema que simula el funcionamiento de una cervecería. El sistema de la cervecería fue desarrollado por los 4 integrantes del grupo, divididos en dos subgrupos, los cuales intercambiaron los bloques de códigos entre ellos para realizar el respectivo testing aplicando las siguientes pruebas:

- **Pruebas de caja negra:** Basadas en verificar si el código cumple los contratos especificados, sin observar el código fuente que resuelve el problema
- **Pruebas de caja blanca:** Se analiza el código fuente, prestando atención a los posibles caminos que puede tomar el código por las estructuras de decisión.
- **Prueba de persistencia:** Se observará si los datos de la cervecería se almacenan y se recuperan correctamente. En el caso de este trabajo se utiliza persistencia binaria
- **Prueba de Interfaz Gráfica:** También llamado Test de GUI, consiste en revisar la parte gráfica del proyecto, específicamente en este trabajo se analiza el módulo de ventana login a la cervecería.
- **Prueba de Integración:** Diseñadas para probar la interacción entre los distintos componentes de un sistema.

# Caja Negra

Para empezar el testeo del trabajo hemos comenzado determinando escenarios, tabla de particiones y batería de prueba para cada método de beerHouse.

En el trabajo todos los métodos que son de relevancia para la SRS han sido testeados dentro del paquete testCajaNegra utilizando la herramienta JUnit obteniendo los siguientes resultados:



A modo de ejemplo se muestran a continuación los escenarios, tablas de particiones y batería de pruebas para algunos de los métodos sometidos al testeo.

**CLASE BEERHOUSE**

**-public double cerrarMesa(Mesa mesa).**

precondición: la mesa es null o la mesa existe y su estado es Ocupada.

**Escenarios**:

| Nro Escenarios | Descripción |
|---|---|
| 1 | mesa existe y está ocupada |
| 2 | mesa null |

**Tabla de particiones:**

| Condición de entrada | Clases válidas | Clases Inválidas |
|---|---|---|
| mesa | mesa válida seleccionada (1), | mesa=null (2) |
| | mesa con comanda valida (3) | mesa con comanda null(4) |

**Batería de prueba:**

| Tipo de clase(correcta o incorrecta) | Valores de entrada | Salida esperada | Clases de prueba cubiertas |
|---|---|---|---|
| Correcta | Mesa=[ Comanda=[] ] | "Mesa cerrada correctamente" | 1,3 |
| Incorrecta | Mesa=[comanda=null] | "Esta mesa no tiene una comanda asignada" | 1,4 |
| | Mesa=null | "No seleccionaste ninguna mesa" | 2 |

Gracias a las pruebas de caja negra se ha podido comprobar que el método cerrarMesa funciona correctamente y devuelve los resultados esperados.

**CLASE Producto**

**-public Producto(int id, String nombre, double costo, double venta, int stock).**
precondición: número entero , mozo distinto de null.

**Escenarios:**

| Nro Escenarios | Descripción |
|---|---|
| 1 | Creación de un nuevo producto |

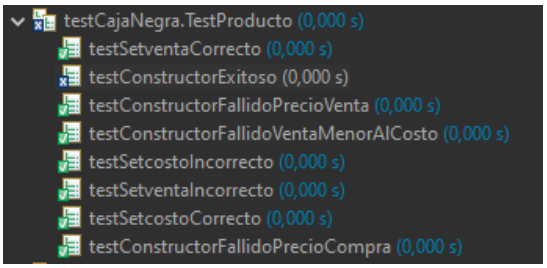
**Tabla de particiones:**

| Condición de entrada | Clases válidas | Clases Inválidas |
|---|---|---|
| costo | costo >0(1) | costo<=0(2) |
| venta | venta>0(3) | venta<=0(4) |
| Estado del objeto | venta>costo(5) | venta<=costo(6) |

**Batería de prueba:**

| Tipo de clase(correcta o incorrecta) | Valores de entrada | Salida esperada | Clases de prueba cubiertas |
|---|---|---|---|
| Correcta | {Id=1,Nombre="Macciato",costo=50,venta=300,stock=100} | Producto agregado satisfactoriamente | 1 - 3 - 5 |
| Incorrecta | {Id=1,Nombre="Macciato",costo=0,venta=300,stock=100} | "Costo INVALIDO , debe ser mayor a 0" | 2 |
| | {Id=1,Nombre="Macciato",costo=50,venta=0,stock=100} | "Precio de venta INVALIDO, debe ser mayor a 0" | 4 |
| | {Id=1,Nombre="Macciato",costo=300,venta=50,stock=100} | "Precio de venta menor al costo" | 6 |

Para cumplir con la SRS se ha probado tanto el constructor como los setters del producto, corroborando con la herramienta JUnit su correcto funcionamiento, el assert lanzado es por un error de assert deprecated.



# Caja Blanca

Agrega Mesa Comanda

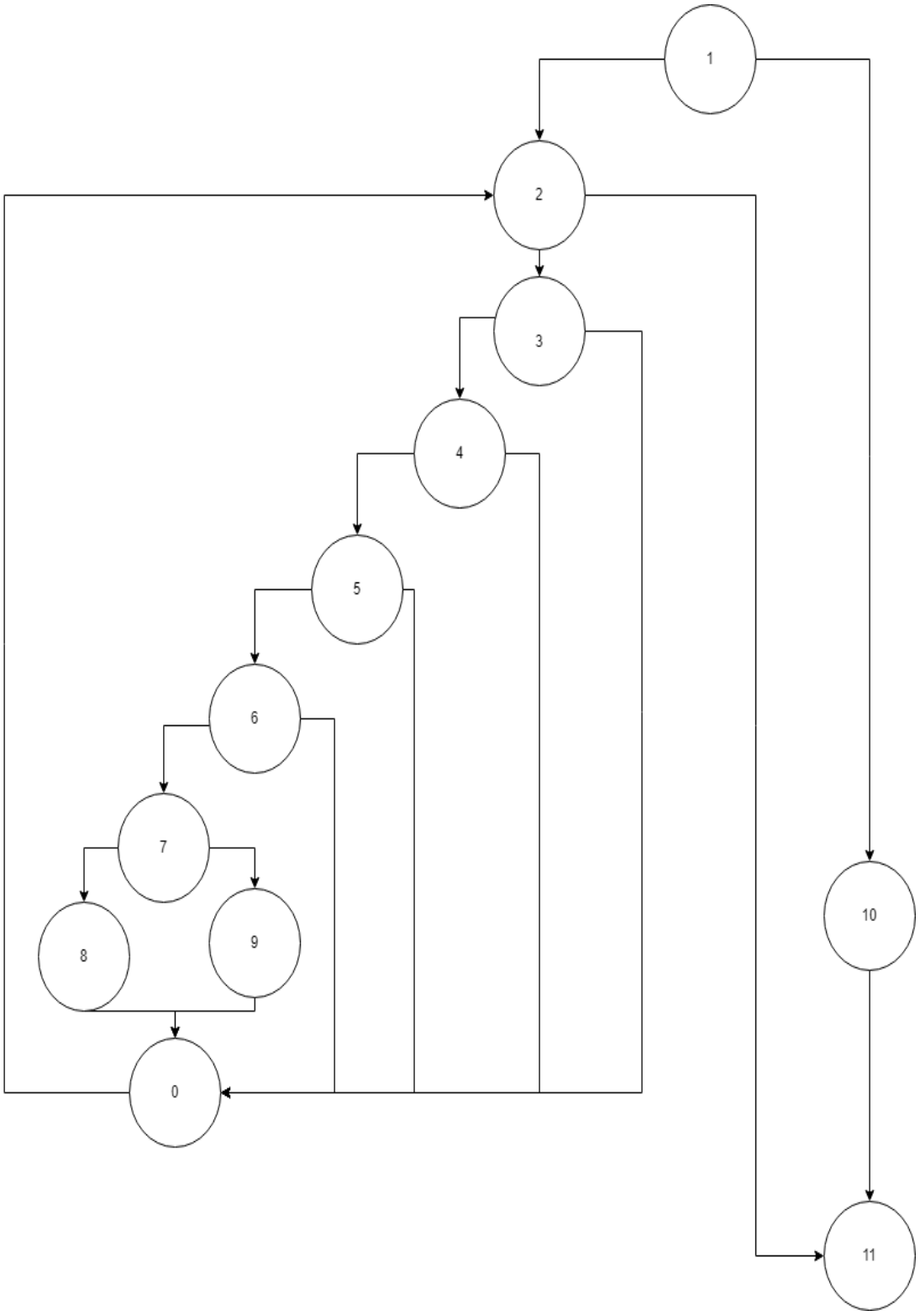
Las pruebas de caja blanca se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. El testeador escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados. Por lo tanto, a diferencia de la caja negra, estas pruebas se centran en analizar el código fuente, intentando que todas las líneas de código se ejecuten.

En el caso particular de este trabajo,vamos a analizar el método “agregaMesaComanda”, el cual le asigna una comanda a una mesa en caso de que se cumplan ciertas condiciones. A continuación adjuntamos una imagen de la implementación de dicho método en código Java.

public void agregaMesaComanda(Pedido pedido,int numeroMesa)

```
boolean search=true;
if(mesa.size()!=0) { //1
    Iterator<Mesa> IteradorMesa = mesa.iterator();
    while(IteradorMesa.hasNext() && search) { //2
        Mesa m = IteradorMesa.next();
        if(m.getNumero()==numeroMesa ) { //3
            if(m.getMozo()!=null) {//4
                if(m.getMozo().getEstado()==0 && m.getEstado().equalsIgnoreCase("libre")) {
                    //5
                    if(verificaPromo(m.getComanda())) {//6
                        if(m.getComanda()==null) { //7
                            Comanda comanda = new Comanda();//8
                            m.setComanda(comanda);
                            m.getComanda().setDate(new GregorianCalendar());
                            m.getComanda().setEstado("abierta");
                        }
                        m.getComanda().addPedido(pedido); //9
                        actualizaStock(pedido.getProducto(), pedido.getCantidad());
                        JOptionPane.showMessageDialog(null, "Mesa: "+ m.getNumero() + "asignada con éxito");
                        search=false;
                    }else
                        throw new MuchosProductosEnPromoException("No pueden haber dos o mas productos en promocion en la misma comanda");
                }else
                    throw new MesaOcupadaException(" Mesa ocupada");
            }else
                throw new  MesaImposibleException("Imposible seleccionar esa mesa");
        }
    }
}else {
    throw new NoHayMesasHabilitadasException(" No hay mesas habilitadas"); //10
} //11
```

## Grafo Ciclomático:



Complejidad ciclomática = arcos - nodos + 2 = 18 - 12 + 2 = **8.**

Por lo tanto tendremos como máximo 8 caminos.

- 1-2-11
- 1-10-11
- 1-2-3-0-11
- 1-2-3-4-0-11
- 1-2-3-4-5-0-11
- 1-2-3-4-5-6-0-11
- 1-2-3-4-5-6-7-8-0-11
- 1-2-3-4-5-6-7-9-0-11

| Camino | Caso | Salida esperada |
|---|---|---|
| 1 | No hay mesas en la cervecería | NoHayMesasHabilitadasException |
| 2 | Camino inaccesible | Camino inaccesible |
| 3 | No está la mesa escogida | - |
| 4 | Mesa sin mozo asignado | MesaImposibleException |
| 5 | La mesa no está libre | MesaOcupadaException |
| 6 | Hay más de un producto en promoción en la comanda | MuchosProductosEnPromoExeption |
| 7 | La comanda de la mesa es nula. | Se le asigna una comanda con estado Abierta a la mesa. Luego se agrega el pedido a dicha comanda. |
| 8 | La comanda de la mesa no es nula. | Se agrega el pedido a la comanda de la mesa. |

Debido a las pruebas de caja blanca realizadas, se pudo corroborar que los resultados obtenidos fueron iguales a los resultados esperados en todos los caminos posibles de nuestro código. Por ende concluimos que la implementación del código es correcta.

# Test de Persistencia

Para el test de persistencia se eligió el módulo Operario. El método de persistencia usado en BeerHouse fue la persistencia binaria.

Durante las pruebas se testean los siguientes escenarios:

- La creación correcta del archivo.
- La escritura en el archivo, tanto con el arreglo de operarios vacío como con datos cargados.
- Despersistir con un archivo incorrecto que no existe.
- Despersistir con el archivo correcto.

En todos estos escenarios no se encontraron fallas ni errores. Por lo que se concluyó que la persistencia del módulo de operarios está bien implementada.

# Test de GUI

Implementamos pruebas para la ventana de login, donde pueden iniciar sesion los operarios, siendo el primero de estos quien queda como Administrador logueandose por única vez con el usuario “ADMIN” contraseña “ADMIN1234” , dándose de alta e ingresando una nueva contraseña a elección sin ningún tipo de verificación, para este ejemplo se utilizó como contraseña para administrador: “pepa”.

El testeo de interfaces gráficas lo hicimos utilizando JUnit 4 y la clase Robot perteneciente al paquete AWT de Java.

El método de testeo automatizado usado consiste en simular la interacción del usuario, utilizando la clase robot para clickear y completar los componentes de la ventana (cuadros de texto, botones, listas, etc).

Como objetivo a testear, nos enfocamos en verificar que los ingresos por teclado estén validados, y que todo funcione como es esperado.

Tras la primer iteración de testeo, nos encontramos los siguientes problemas en el código:

- No se lanza un error ni se valida la nueva contraseña ingresada por el administrador.
- Si los campos de login están vacíos debería tirar una excepción.

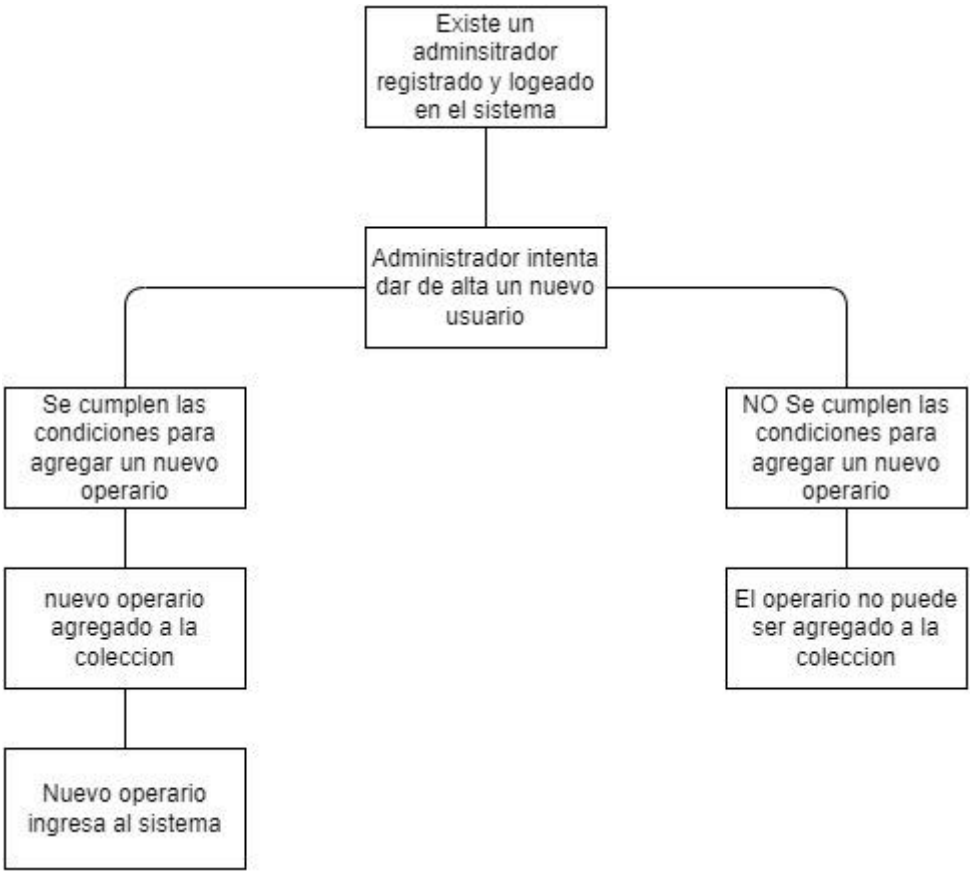
# Test de Integración

**Caso de uso “DarAltaOperario”**:

Descripción: El operario debe ser dado de alta para usar el sistema, utiliza un “username” y una “password” para ello.

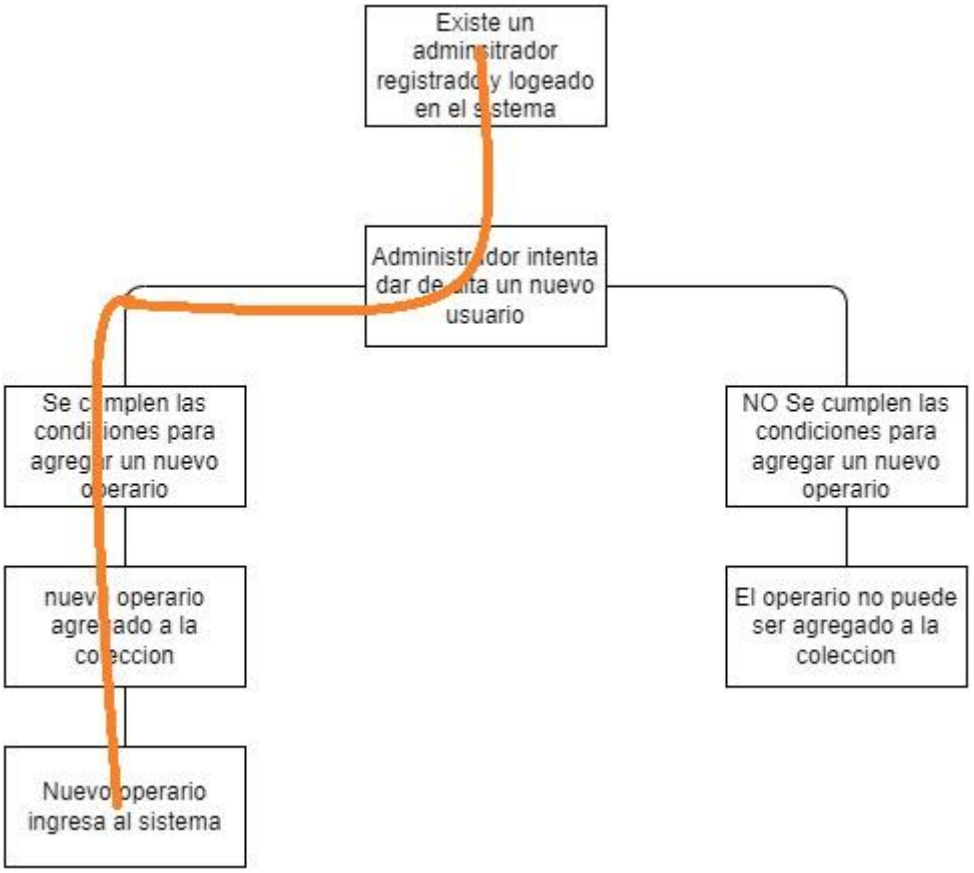
Actores: Operario. Admin.

Pre condiciones: Debe haber un administrador registrado previamente.



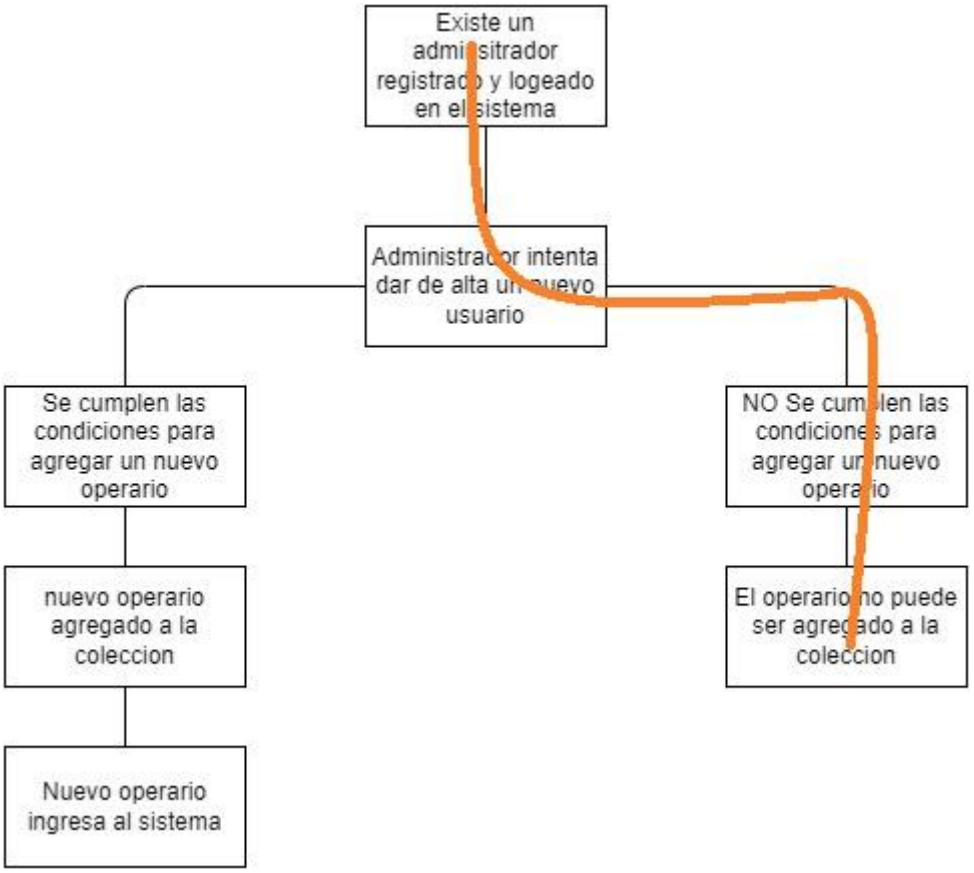

Flujo Normal:

1. El administrador da de alta a un nuevo operario con un “username” y “password”.
2. El operario es agregado a la colección.
3. El operario ingresa al sistema.




Flujo alternativo 1:

1. El admin intenta dar de alta a un nuevo operario con un username y password.
2. El username y password no cumplen con las condiciones.
3. El operario no es agregado a la colección.




Excepciones

a. E1. PasswordInvalidaException.

Post-condiciones: El operario se registra en el sistema.

# Conclusión

A lo largo de este trabajo pudimos implementar diversos métodos de testing. Se detectó mediante el enfrentamiento de códigos de ambos subgrupos cuales eran los errores del sistema de software implementado. También se detectó que por momentos la SRS presentaba algunos fallos de especificación y análisis.

Al momento de hacer las pruebas de caja negra priorizamos a los métodos estructurales de nuestro sistema dado que no es viable realizar caja negra a la totalidad de los métodos.

Utilizamos el Javadoc lo cual nos facilitó la comprensión del código y nos ayudó al realizar las pruebas de caja negra.

Utilizamos las pruebas de caja blanca para verificar el funcionamiento de las líneas de código no ejecutadas hasta el momento. Para eso realizamos el gráfico ciclomático del código. Calculamos la complejidad del mismo, y concluimos haciendo los caminos básicos con su correspondiente salida esperada.

Durante la ejecución de pruebas de persistencia no se encontraron errores. Se eligió elegir las clases Operario.

Durante la ejecución del test de GUI no todos los casos se ejecutaron de la mejor manera debido a los JOptionPane y además un caso no se probó correctamente debido a la persistencia.