SISTEMA DE CERVECERÍA

TALLER DE PROGRAMACIÓN I
UNMDP

SUBGRUPO

-RODRÍGUEZ, JUAN GABRIEL
-UZQUIANO, TOMÁS EMANUEL

INTEGRANTES

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

ÍNDICE

Introducción

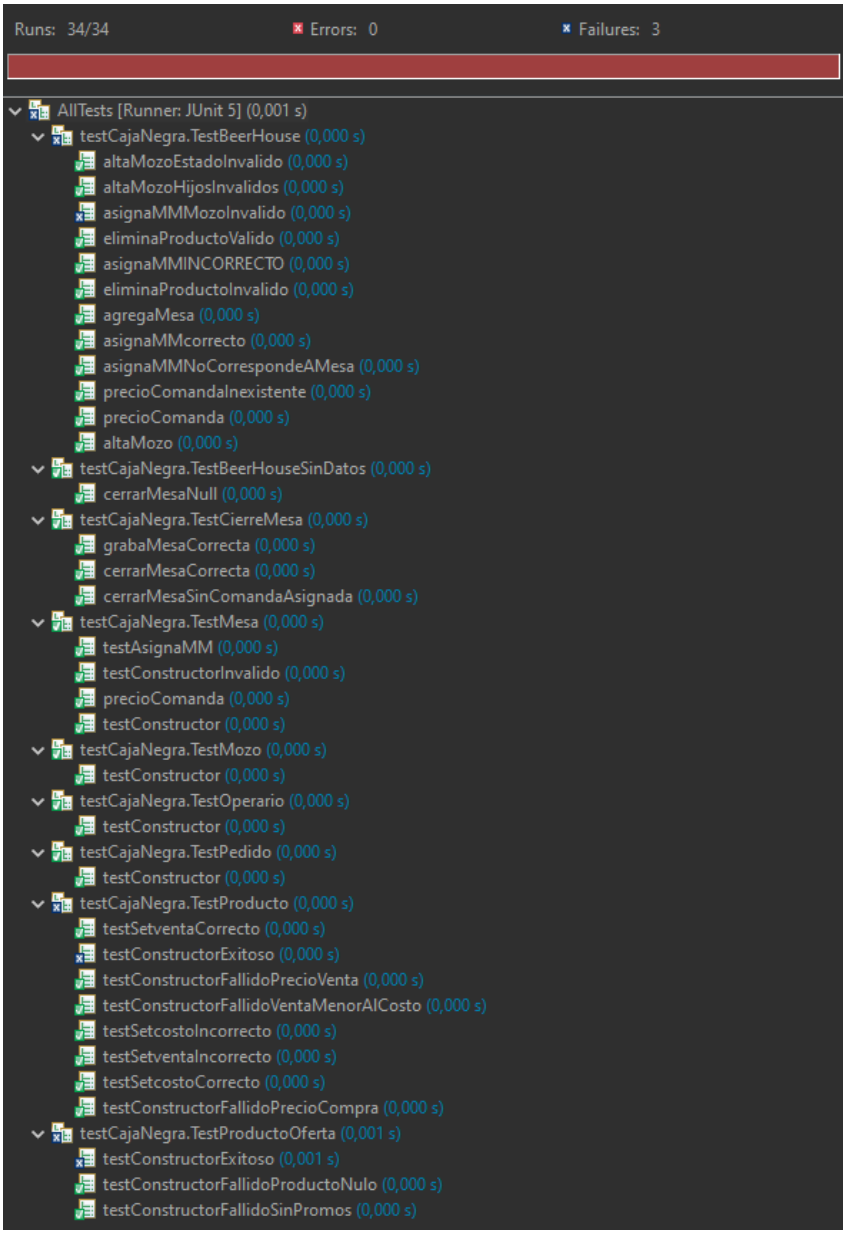
En el presente informe se detallarán los resultados obtenidos tras aplicar técnicas de testing en un sistema que simula el funcionamiento de una cervecería. El sistema de la cervecería fue desarrollado por los 4 integrantes del grupo, divididos en dos subgrupos, los cuales intercambiaron los bloques 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 precioComanda(Mesa mesa).

precondición: comanda distinta de null, mesa distinta de null, pedido distinto de null.

Escenarios:

Nro Escenarios	Descripción
1	Colección de mesas con una mesa n°2

Tabla de particiones:

Condición de entrada	Clases válidas	Clases Inválidas
mesa	mesa en colección "ocupada" con comanda válida(1)	mesa en colección sin comanda(2)

Batería de prueba:

Tipo de clase(correcta o incorrecta)	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Correcta	{numero=2; 4; Mozo; "ocupada"; Comanda; 0, 0}	Precio total con descuentos	1 - 3
Incorrecta	{numero=2; 4; Mozo; "ocupada"; null; 0, 0}	Mesa sin comanda asignada	2

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

CLASE BEERHOUSE

-public void asignaMM(int numero,Mozo mozo).

precondición: numero entero.

Escenarios:

Nro Escenarios	Descripción
1	Colección con al menos un mozo llamado Jose, y mesa n°2 habilitada y libre
2	Colección sin mozo, y mesa n°2 habilitada y libre

Tabla de particiones:

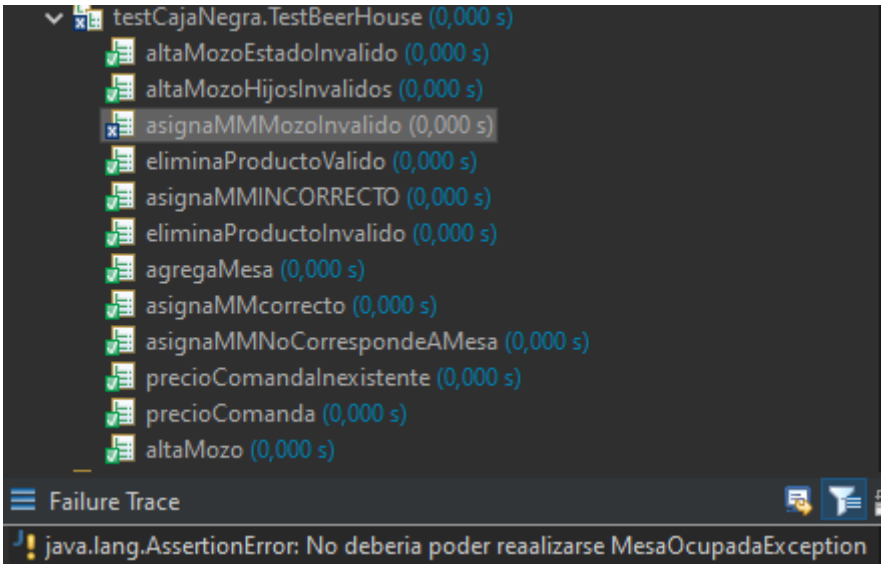
Condición de entrada	Clases válidas	Clases Inválidas
numero	numero >= 0 que pertenece a una mesa habilitada y libre (1)	numero que no corresponde a una mesa (2)
mozo	mozo valido (3)	mozo null (4)
Estado de colección	Existe el mozo en la colección (5)	No existe el mozo en la colección (6)

Batería de prueba:

Tipo de clase(correcta o incorrecta)	Valores de entrada	Salida esperada	Clases de prueba cubiertas
Correcta	{numero=2; [mozo=Jose;02/21/1980;1;1;5]; escenario 1}	Mesa asignada con Exito!	1 - 3 - 5
Incorrecta	{numero = 999 ; [mozo=Jose;02/21/1980;1;1;5]; escenario 1}	"Ese numero no corresponde a una mesa"	2
Incorrecta	{numero = 999 ;	"Debes seleccionar un	6

	[mozo=Marcos;22/10/1999 ;3;1;1]; escenario 1}	mozo valido de la lista"	
Incorrecta	{numero = 999 ; [mozo=null]; escenario 2}	Mesa asignada con Exito!	4

Gracias al uso adecuado de asertos hemos podido detectar que el método analizado mediante el test de caja negra no funciona correctamente.



En dicho método de prueba se asigna a una mesa un mozo, el cual ingresa como null , el método permite esto y devuelve el cartel al usuario “Mesa asignada con Exito!” cuando no debería suceder.

Caja Blanca

eliminaProducto

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.

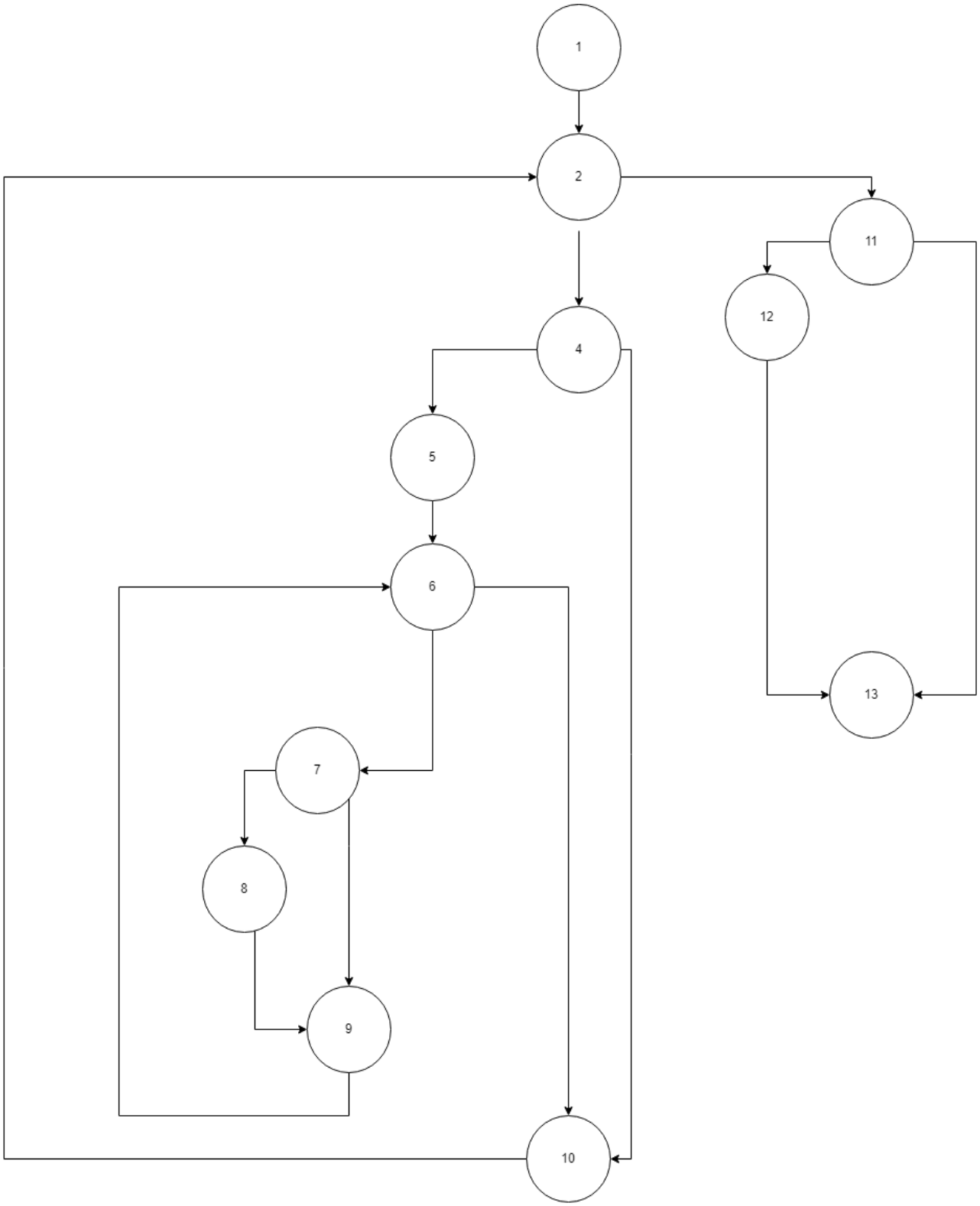
En el caso particular de este trabajo,vamos a analizar el método “eliminaProducto”, teniendo en cuenta que la eliminación del producto no puede efectuarse si está asociado a una comanda. A continuación adjuntamos una imagen de la implementación de dicho método en código Java.

public void eliminaProducto(Producto producto)

```
public void eliminaProducto(Producto producto)  throws ProductoAsociadoAComandaException{
    boolean ok=true; //1
    Iterator<Mesa> Iterador = mesa.iterator();
    while(Iterador.hasNext() && ok) { //2
        Mesa m = Iterador.next(); //3
        System.out.println(m.toString());
        Comanda comanda= m.getComanda();
        if(comanda!=null) { //4
            ArrayList<Pedido> pedidos= comanda.getOrden(); //5
            Iterator<Pedido> IteradorPed = pedidos.iterator();
            while(IteradorPed.hasNext() && ok) { //6
                Pedido p = IteradorPed.next();
                if(pedidos!=null && p.getProducto()==producto ) { //7
                    ok=false; //8
                    throw new ProductoAsociadoAComandaException("No se puede eliminar el producto, pertenece a una comanda");
                    }
                } //9
            }
    } //10
    if(ok) { //11
        this.producto.remove(producto); //12
    }

} //13
```

Grafo ciclomático



Complejidad ciclomática = arcos - nodos + 2 = 16 - 12 + 2 = **6.**

Por lo tanto tendremos como máximo **6** caminos.

1. 1-2-11-13
2. 1-2-11-12-13
3. 1-2-4-10-11-12-13
4. 1-2-4-5-6-10-2-11-12-13
5. 1-2-4-5-6-7-9-10-11-12-13
6. 1-2-4-5-6-7-8-9-10-11-12-13

Camino	Caso	Salida esperada
1	Camino imposible	Camino imposible
2	No hay mesas	Se elimina el producto
3	Hay una mesa y no tiene comanda	Se elimina el producto
4	Hay mesa con comanda pero sin pedidos asociados a ella	Se elimina el producto
5	Hay mesa con comanda pero no tiene asociado el producto	Se elimina el producto
6	Hay una mesa con el pedido asociado a su comanda	ProductoAsociadoAComandaException

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 Mesa. 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 mesas vacías 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 mesas 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 “Login”:

Descripción: El operario (ya sea el administrador o uno corriente) debe identificarse con un “username” y una “password” para poder acceder al sistema.

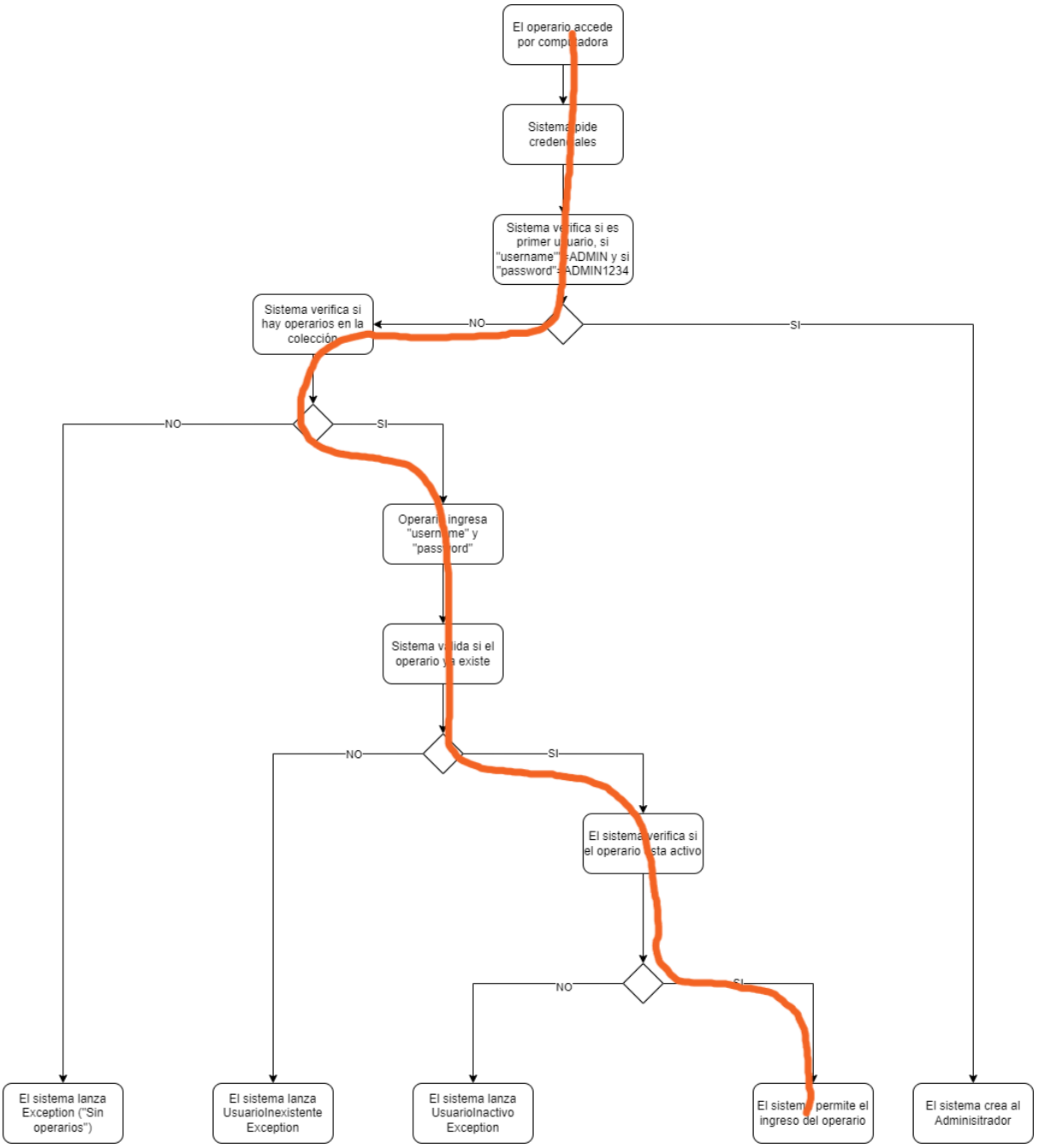
Flujo Normal 1:

1. El operario ingresa su “username” y su “password”.
2. Si los datos coinciden el operario ingresará al sistema.

Actores: Operario.

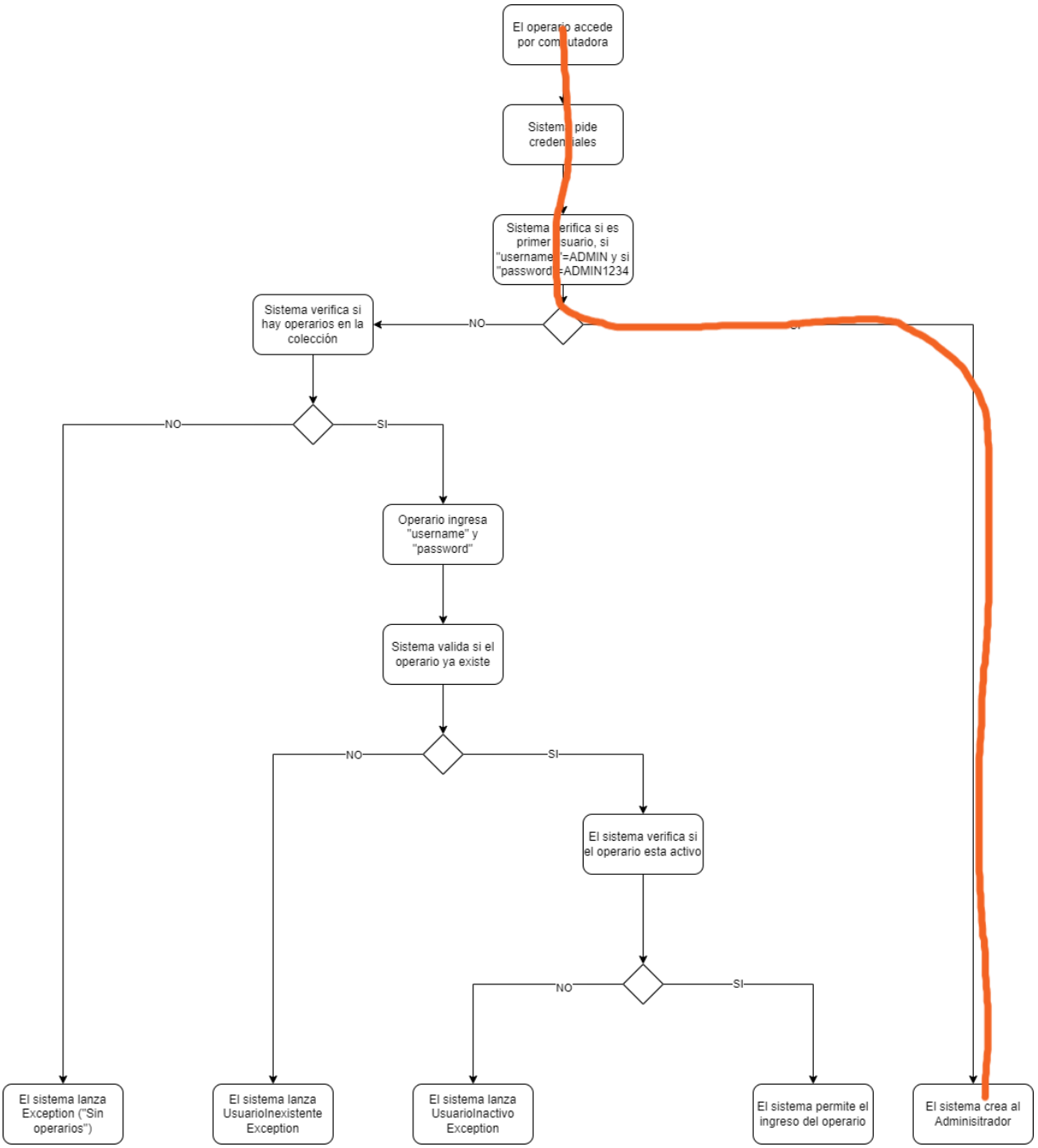
Pre condiciones:

- Username no es vacío ni null.
- Password no es vacío ni null.


El operario accede por computadora
Sistema pide credenciales
Sistema verifica si es primer usuario, si "username"=ADMIN y si "password"=ADMIN1234
NO
SI
Sistema verifica si hay operarios en la colección
NO
SI
Operario ingresa "username" y "password"
Sistema valida si el operario ya existe
NO
SI
El sistema verifica si el operario esta activo
NO
SI
El sistema lanza Exception ("Sin operarios")
El sistema lanza UsuarioInexistente Exception
El sistema lanza UsuarioInactivo Exception
El sistema permite el ingreso del operario
El sistema crea al Adminisitrador

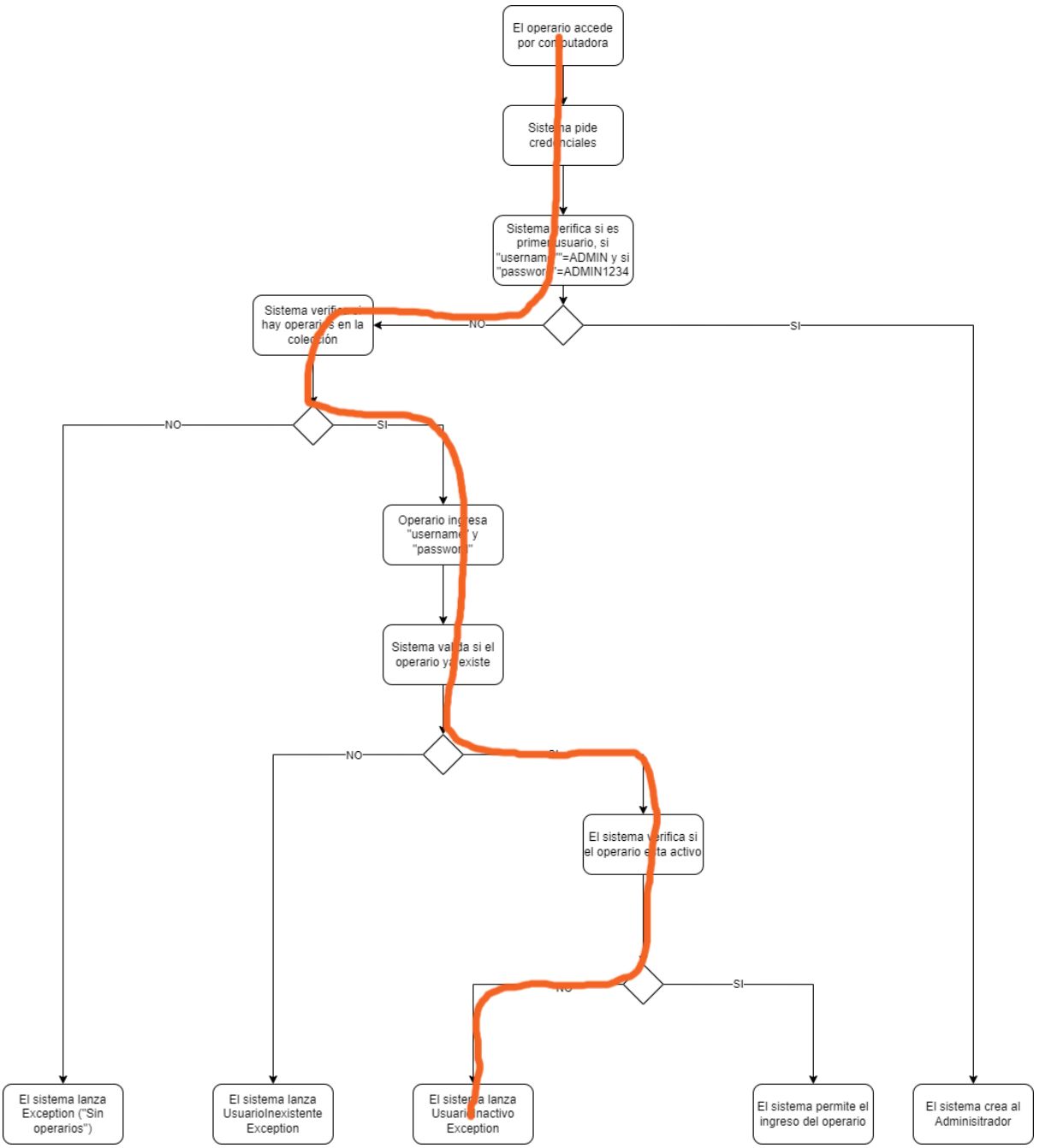
Flujo alternativo 1:

1. El operario ingresa su “username”=ADMIN y su “password”=ADMIN1234 y la colección de operarios está vacía.
2. El operario toma el rol de administrador.



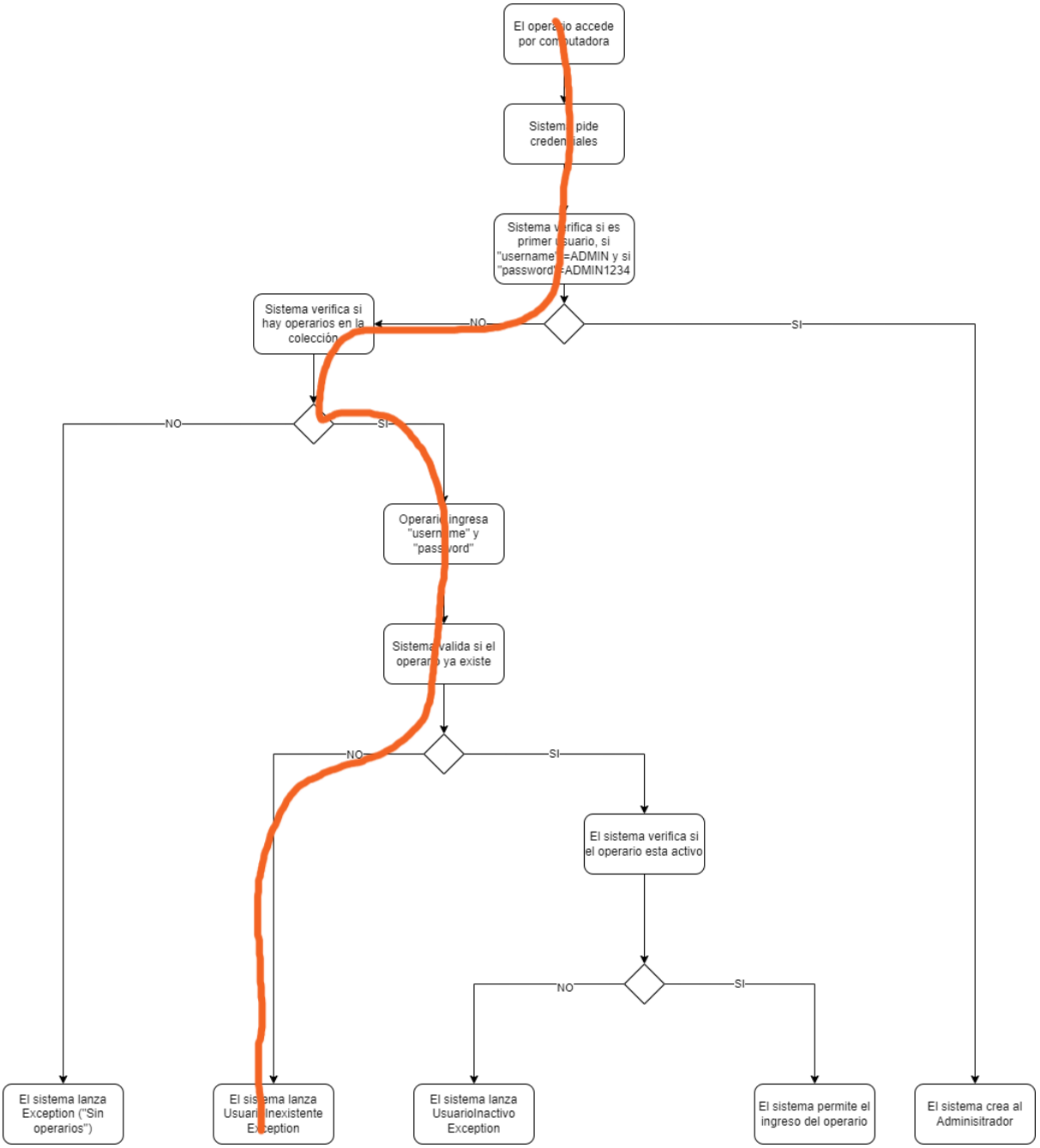
Flujo alternativo 2:

1. El operario ingresa su “username”!=ADMIN y su “password”!=ADMIN1234.
2. La colección de operarios no está vacía.
3. Los datos coinciden pero el operario está inactivo.
4. El operario existe en el sistema.
5. El sistema verifica que el operario está inactivo.
6. El sistema tira UsuarioInactivo excepción.



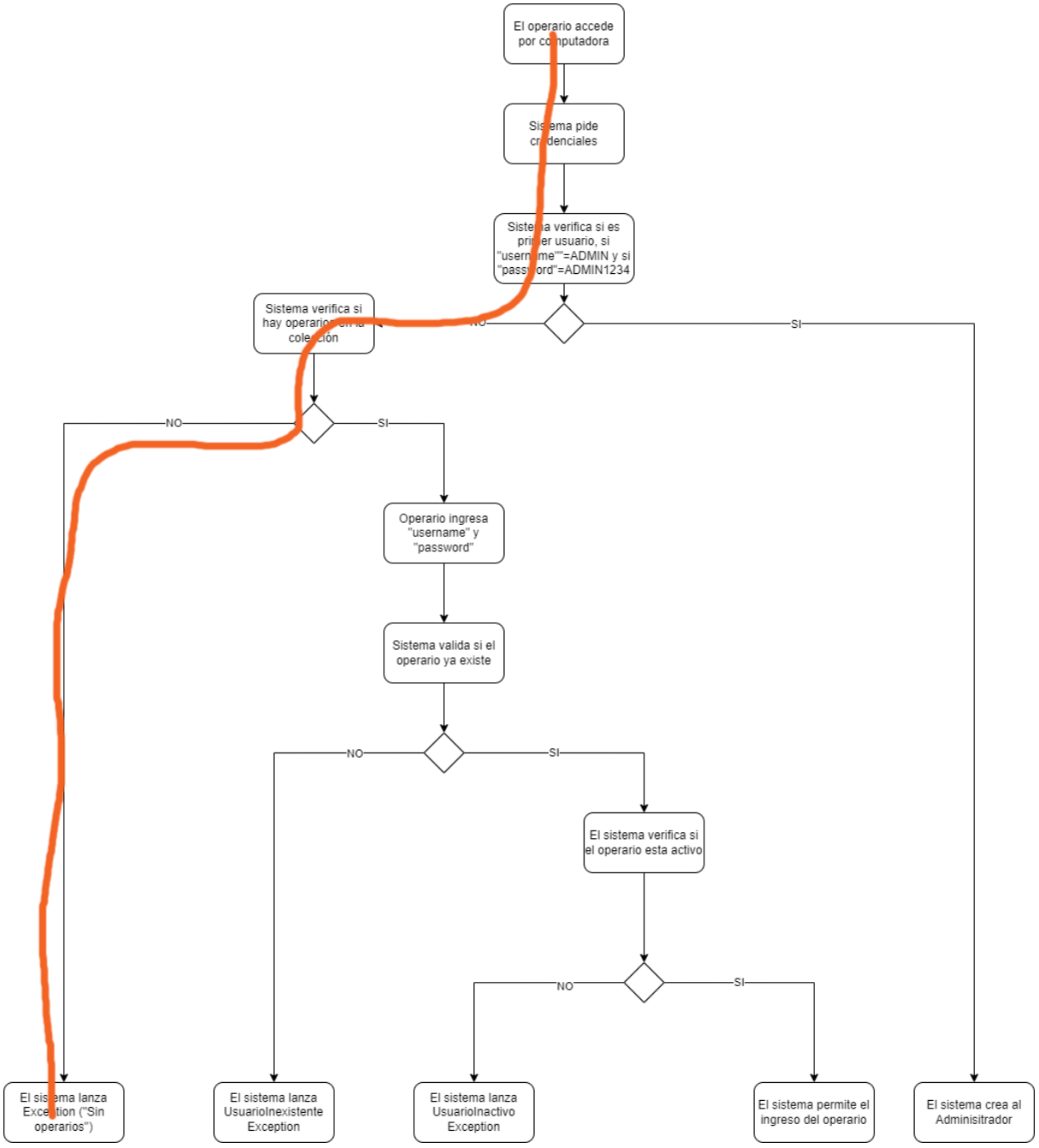
Flujo alternativo 3:

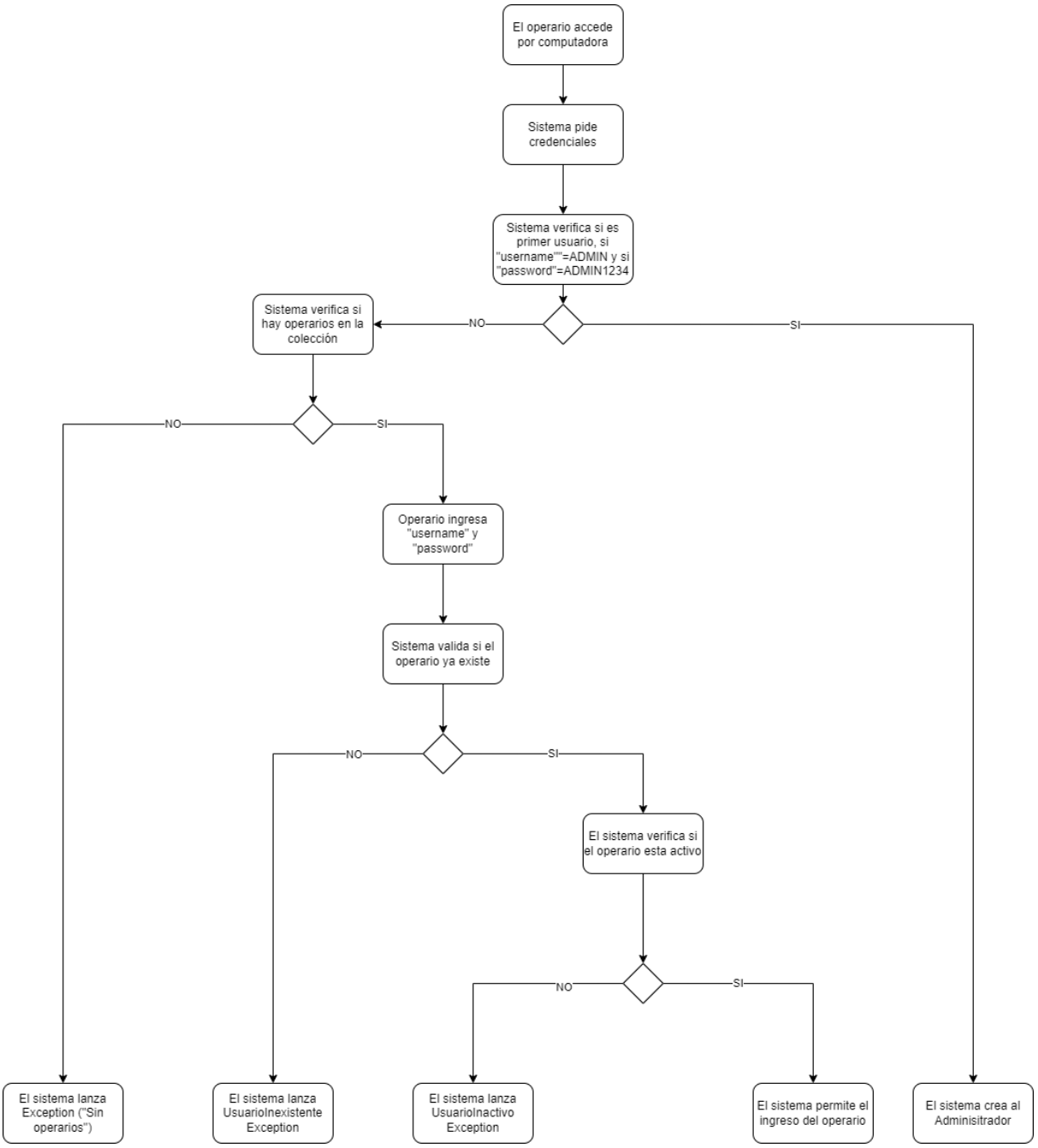
1. El operario ingresa su “username”!=ADMIN y su “password”!=ADMIN1234.
2. La colección de operarios está vacía.
3. El operario no existe en el sistema.
4. El sistema tira UsuarioInexistente excepción.



Flujo alternativo 4:

1. El operario ingresa su “username”!=ADMIN y su “password”!=ADMIN1234.
2. La colección de operarios está vacía.
3. El sistema tira Exception excepción.





Excepciones

a. E1. UsuarioInactivoException.
b. E2. UsuarioInexistenteException.
c. E3. Exception.

Post-condiciones: El operario ingresa al 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 Mesa.

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.