



SISTEMA DE CERVECERÍA

TALLER DE PROGRAMACIÓN I
UNMDP

SUBGRUPO

-BOOLLS, NICOLÁS
-GUTIÉRREZ, ALAN

INTEGRANTES

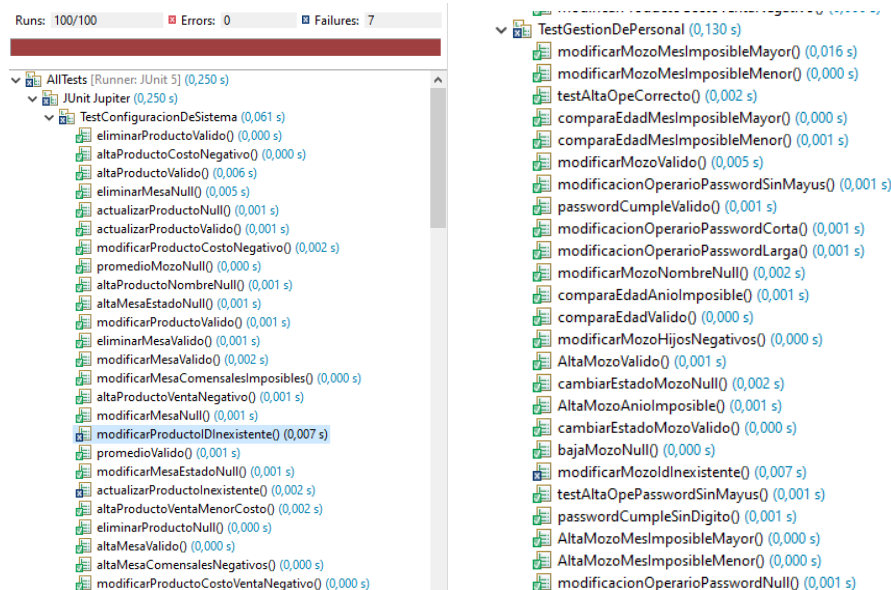
-BOOLLS, NICOLÁS
-GUTIÉRREZ, ALAN
-RODRÍGUEZ, JUAN GABRIEL
-UZQUIANO, TOMÁS EMANUEL

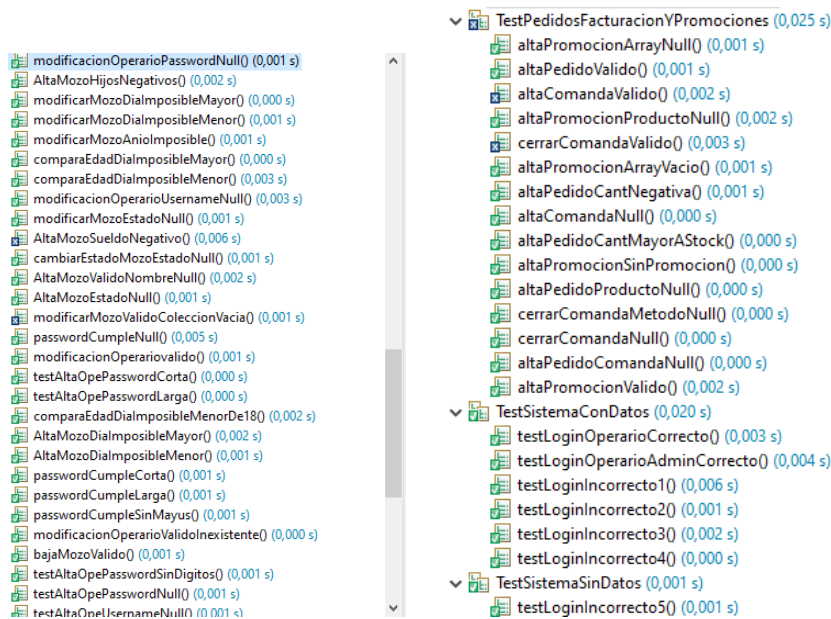
Introducción

Caja Negra

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

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





Concluimos que en términos generales los métodos funcionan correctamente excepto en el caso particular de que se quiera modificar un elemento de una colección vacía o un elemento inexistente. La mayor parte de los errores encontrados fueron por estos casos. Por ejemplo, en el caso particular de que se quiera modificar los atributos de un mozo con un ID inexistente, el método nos devuelve una excepción que nos dice que se está intentando acceder a un índice imposible de la colección.

Considero que estos errores fueron más por un descuido de nuestros compañeros a la hora de hacer el contrato que por errores de codificación, ya que después de analizar el funcionamiento del programa, me parece que estos casos deberían estar contemplados en las precondiciones.

Luego, a modo de observación, testeamos el caso en el que el sueldo de un mozo es negativo. Observamos que en caso de que esto se cumpla, el programa continúa su ejecución normal. A pesar de no estar en el contrato, me pareció interesante para el programador tener en cuenta este detalle, que podría ser tanto un descuido en el armado del contrato, como una característica del programa.

Adjunto a continuación el link de Google Sheets que contiene todos los escenarios, tablas de particiones y batería de pruebas para algunos de los métodos sometidos al testeo.

[Caja Negra - Boolls, Gutierrez](#)

Test de cobertura

Una vez hechos los tests de cobertura, escogimos los métodos modificarMesa, altaMesa y altaComanda para aplicarles el test de caja blanca, ya que nos parecieron los métodos más relevantes con mayor cantidad de líneas sin ejecutar. Adjunto imágenes a continuación:

```
public void modificarMesa(Mesa mesa, int cantComensales, String estado) throws DatosIncorrectosException {
    if( mesa==null || estado==null) //1
        throw new DatosIncorrectosException("Parametros invalidos.");
    if(mesa.getId() >= 1) { //2
        if(cantComensales >= 2) { //3
            int i = 0;
            ArrayList<Mesa> mesas = Sistema.getInstance().getMesas();
            while(i < mesas.size() && mesas.get(i).getId() != mesa.getId()) //4
                i++; //5
            mesas.get(i).setCantComensales(cantComensales);
            mesas.get(i).setEstado(estado);
        } //6
        else //7
            throw new DatosIncorrectosException("Cantidad de comensales debe ser mayor a 1.");
    } //8
    else{ //9
        int i = 0;
        ArrayList<Mesa> mesas = Sistema.getInstance().getMesas();
        while(i < mesas.size() && mesas.get(i).getId() != mesa.getId()) //10
            i++; //11
        mesas.get(i).setCantComensales(cantComensales);
        mesas.get(i).setEstado(estado);
    } //12
} //13

*/
public void altaMesa(int cantComensales, String estado) throws DatosIncorrectosException {
    if(estado==null)
        throw new DatosIncorrectosException("Estado nulo.");
    if(Sistema.getInstance().getMesas().size() >= 1) {
        if(cantComensales >= 2) {
            Sistema.getInstance().getMesas().add(new Mesa(cantComensales, estado));
        }
        else
            throw new DatosIncorrectosException("Cantidad de comensales debe ser mayor a 1.");
    }
    else
        Sistema.getInstance().getMesas().add(new Mesa(cantComensales, estado));
}
```

```

60 public void altaComanda(Comanda comanda) throws DatosIncorrectosException {
61     Mozo mozoAsociado;
62     int i=0;
63     boolean hayMesaLibre=false;
64     Producto producto;
65     if(comanda==null) //1
66         throw new DatosIncorrectosException("Parametros invalidos."); //2
67     for(Mesa mesa: Sistema.getInstance().getMesas()) { //3
68         if(mesa.getEstado().equals(EstadosMesa.LIBRE.getEstado())){ //4
69             hayMesaLibre=true; //4.1
70         } //5
71     } //6
72     if(!hayMesaLibre) //7
73         throw new DatosIncorrectosException("No hay mesas libres en el restaurante"); //8
74
75     if(comanda.getMesa().getEstado().equals(EstadosMesa.OCUPADA.getEstado()) ) //9
76         throw new DatosIncorrectosException("La mesa esta ocupada, debe elegir una mesa activa."); //10
77
78     mozoAsociado =Sistema.getInstance().getAsignacionesDesdeMesa().get(comanda.getMesa().getId()).getMozo()
79
80     if( mozoAsociado==null || !mozoAsociado.getEstado().equals(Estado.ACTIVO.getEstado())) //11
81         throw new DatosIncorrectosException("El mozo asociado no existe o no se encuentra disponible."); //
82
83     /*De aca para abajo controla que haya al menos dos promociones activas*/
84     ArrayList<Promocion> promociones=Sistema.getInstance().getPromociones();
85     if(promociones==null) //13
86         throw new DatosIncorrectosException("No hay promociones activas."); //14
87
88     while(i<promociones.size() && !promociones.get(i).isActiva()) { //15     la condicion aqui estaba mal
89         i++; //16
90     }
91     if(i<promociones.size()) //17
92         producto=promociones.get(i).getProducto(); //18
93     else //19
94         throw new DatosIncorrectosException("No hay promociones activas."); //20
95
96     i++;
97     while(i<promociones.size() && !promociones.get(i).isActiva() && producto.getId()!=promociones.get(i).ge
98         i++; //22
99     if(!(i<promociones.size())) //23
100         throw new DatosIncorrectosException("No hay suficientes promociones activas. Se necesitan al menos
101
102     ArrayList<Mesa> mesas= Sistema.getInstance().getMesas();
103     for(Mesa mesa: mesas) { //25
104         if(comanda.getMesa().getId()== mesa.getId()) //26
105             mesa.setEstado(EstadosMesa.OCUPADA.getEstado()); //26.1
106     } //27
107
108
109     Sistema.getInstance().getComandas().add(comanda);
110
111
112 } //28
113

```

Caja Blanca

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

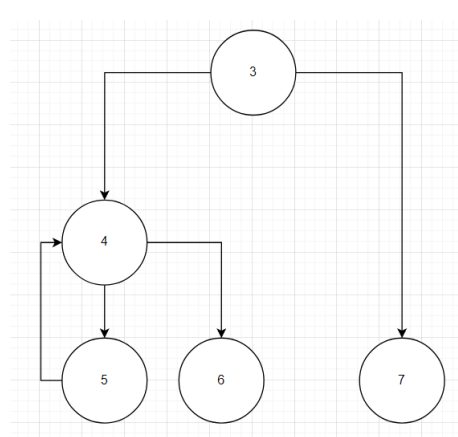
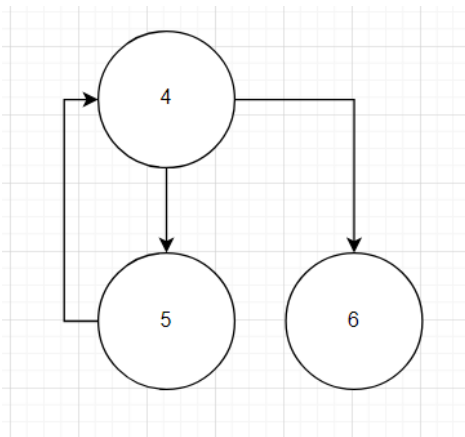
adecuados. Por lo tanto, a diferencia de la caja negra, estas pruebas se centran en analizar el código fuente, intentando que todas las líneas de código se ejecuten.

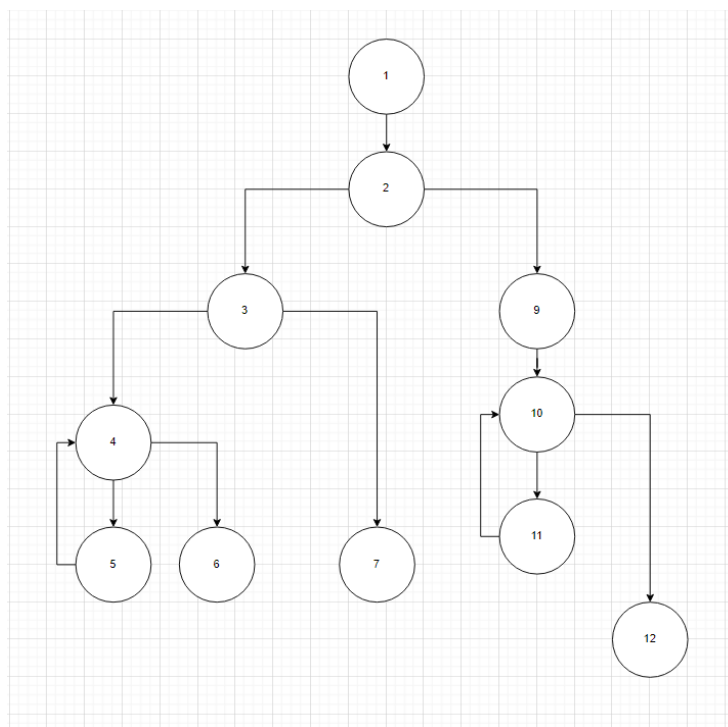
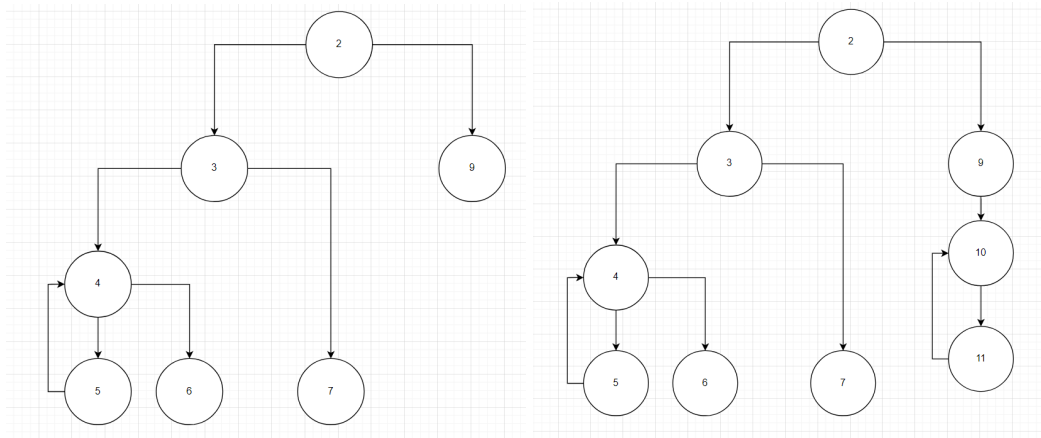
En el caso particular de este trabajo, vamos a analizar los métodos mencionados anteriormente.

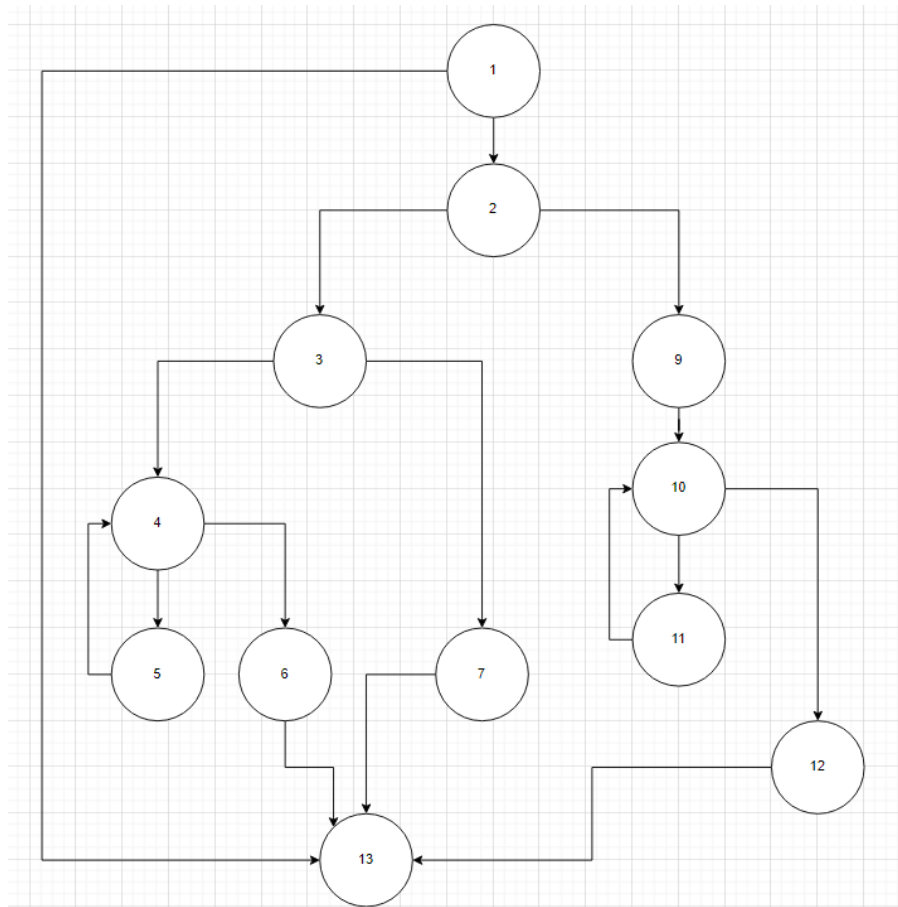
modificarMesa

```
public void modificarMesa(Mesa mesa, int cantComensales, String estado) throws DatosIncorrectosException {
    if( mesa==null || estado==null) //1
        throw new DatosIncorrectosException("Parametros invalidos.");
    if(mesa.getId() >= 1) { //2
        if(cantComensales >= 2) { //3
            int i = 0;
            ArrayList<Mesa> mesas = Sistema.getInstance().getMesas();
            while(i < mesas.size() && mesas.get(i).getId() != mesa.getId()) //4
                i++; //5
            mesas.get(i).setCantComensales(cantComensales);
            mesas.get(i).setEstado(estado);
        } //6
        else //7
            throw new DatosIncorrectosException("Cantidad de comensales debe ser mayor a 1.");
    } //8
    else{ //9
        int i = 0;
        ArrayList<Mesa> mesas = Sistema.getInstance().getMesas();
        while(i < mesas.size() && mesas.get(i).getId() != mesa.getId()) //10
            i++; //11
        mesas.get(i).setCantComensales(cantComensales);
        mesas.get(i).setEstado(estado);
    } //12
} //13
```

Para la realización del grafo cicломático utilizamos el método ascendente.







Complejidad ciclomática = Regiones + región exterior = 5 + 1 = 6

Por ende, tendremos como máximo 6 caminos.

Para determinarlos, utilizamos el método simplificado:

Camino 1: 1-13

Camino 2: 1-2-3-7-13

Camino 3: 1-2-3-4-6-13

Camino 4: 1-2-9-10-12-13

Camino 5: 1-2-3-4-5-4-6-13

Camino 6: 1-2-9-10-11-10-12-13

| Camino | Caso | Salida esperada |
|--------|---------------------------------|---------------------------|
| 1 | mesa=null estado=null | DatosIncorrectosException |
| 2 | La mesa tiene id mayor a 1 pero | DatosIncorrectosException |

| | | |
|---|---|---------------------|
| | cantidad de comensales < 2 | |
| 3 | La mesa tiene id mayor a 1 y es la primera de la colección | Se modifica la mesa |
| 4 | La mesa tiene id < 1 y es la primera de la colección | Se modifica la mesa |
| 5 | La mesa tiene id mayor a 1 y no es la primera de la colección | Se modifica la mesa |
| 6 | La mesa tiene id < 1 y no es la primera de la colección | Se modifica la mesa |

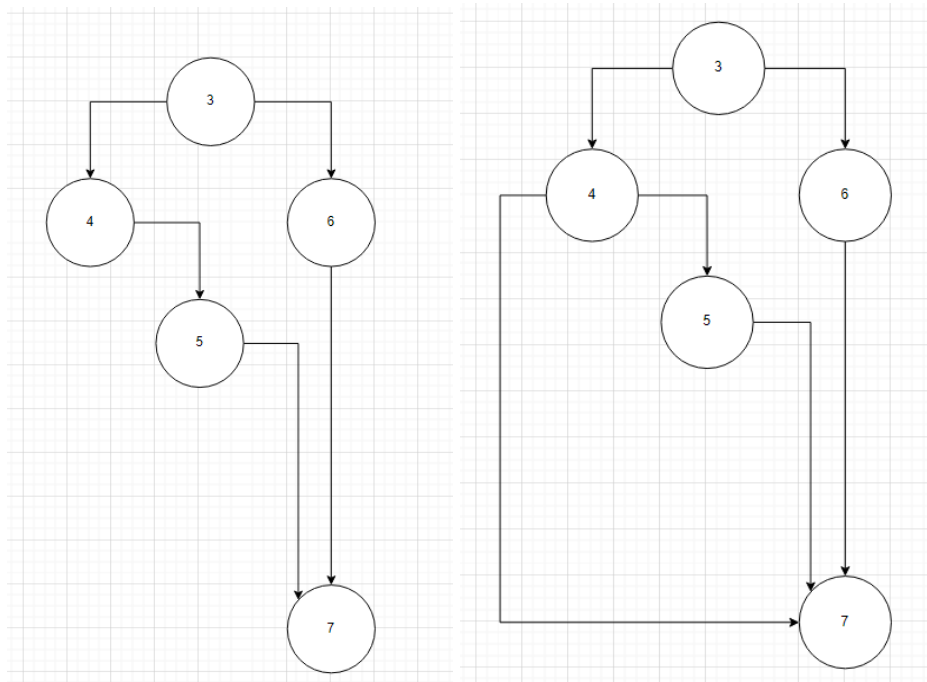
altaMesa

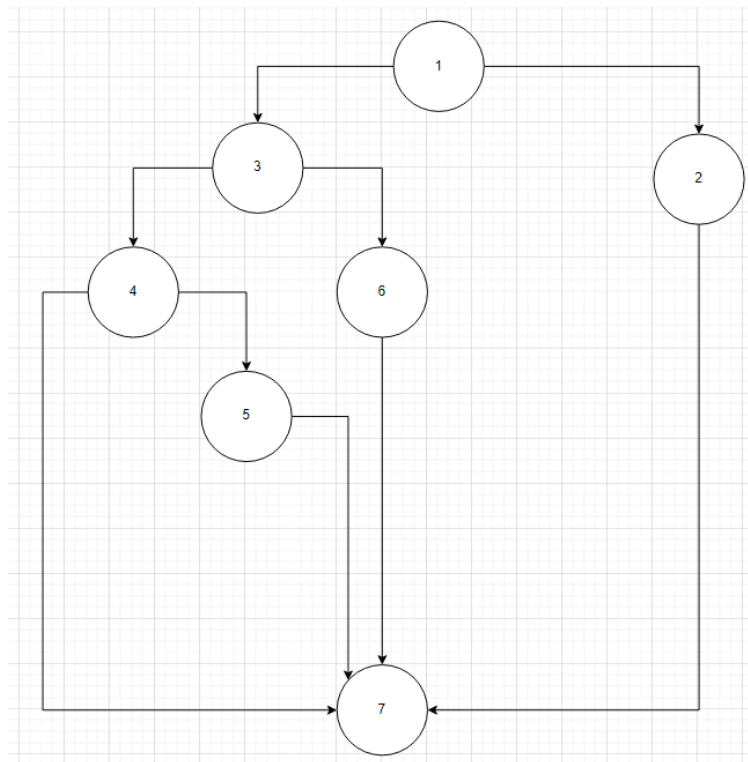
```

public void altaMesa(int cantComensales, String estado) throws DatosIncorrectosException {
    if(estado==null) //1
        throw new DatosIncorrectosException("Estado nulo."); //2
    if(Sistema.getInstance().getMesas().size() >= 1) { //3
        if(cantComensales >= 2) { //4
            Sistema.getInstance().getMesas().add(new Mesa(cantComensales,estado));
        }
        else //5
            throw new DatosIncorrectosException("Cantidad de comensales debe ser mayor a 1.");
    }
    else //6
        Sistema.getInstance().getMesas().add(new Mesa(cantComensales,estado));
} //7

```

Para la realización del grafo ciclomático utilizamos el método ascendente.





Complejidad ciclomática = Regiones + región exterior = 3 + 1 = 4

Por ende, tendremos como máximo 4 caminos.

Para determinarlos, utilizamos el método simplificado:

Camino 1: 1-2-7

Camino 2: 1-3-6-7

Camino 3: 1-3-4-7

Camino 4: 1-3-4-5-7

| Camino | Caso | Salida esperada |
|--------|--|---------------------------|
| 1 | estado=null | DatosIncorrectosException |
| 2 | Coleccion de mesas vacía | Agrega la mesa |
| 3 | Colección de mesas con al menos un elemento y cantComensales >=2 | Se agrega la mesa |
| 4 | Colección de mesas con al menos | DatosIncorrectosException |

| | | |
|--|----------------------------------|--|
| | un elemento y cantComensales < 2 | |
|--|----------------------------------|--|

altaComanda

```

public void altaComanda(Comanda comanda) throws DatosIncorrectosException {
    Mozo mozoAsociado;
    int i=0;
    boolean hayMesaLibre=false;
    Producto producto;
    if(comanda==null) //1
        throw new DatosIncorrectosException("Parametros invalidos."); //2
    for(Mesa mesa: Sistema.getInstance().getMesas()) { //3
        if(mesa.getEstado().equals(EstadosMesa.LIBRE.getEstado())){ //4
            hayMesaLibre=true; //4.1
        } //5
    } //6
    if(!hayMesaLibre) //7
        throw new DatosIncorrectosException("No hay mesas libres en el restaurante"); //8

    if(comanda.getMesa().getEstado().equals(EstadosMesa.OCUPADA.getEstado()) ) //9
        throw new DatosIncorrectosException("La mesa esta ocupada, debe elegir una mesa activa."); //10

    mozoAsociado =Sistema.getInstance().getAsignacionesDesdeMesa().get(comanda.getMesa().getId()).getMozo();

    if( mozoAsociado==null || !mozoAsociado.getEstado().equals(Estado.ACTIVO.getEstado())) //11
        throw new DatosIncorrectosException("El mozo asociado no existe o no se encuentra disponible."); //12

    /*De aca para abajo controla que haya al menos dos promociones activas*/
    ArrayList<Promocion> promociones=Sistema.getInstance().getPromociones();
    if(promociones==null) //13
        throw new DatosIncorrectosException("No hay promociones activas."); //14

    while(i<promociones.size() && !promociones.get(i).isActiva()) { //15     la condicion aqui estaba mal
        i++; //16
    }

    if(i<promociones.size()) //17
        producto=promociones.get(i).getProducto(); //18
    else //19
        throw new DatosIncorrectosException("No hay promociones activas."); //20

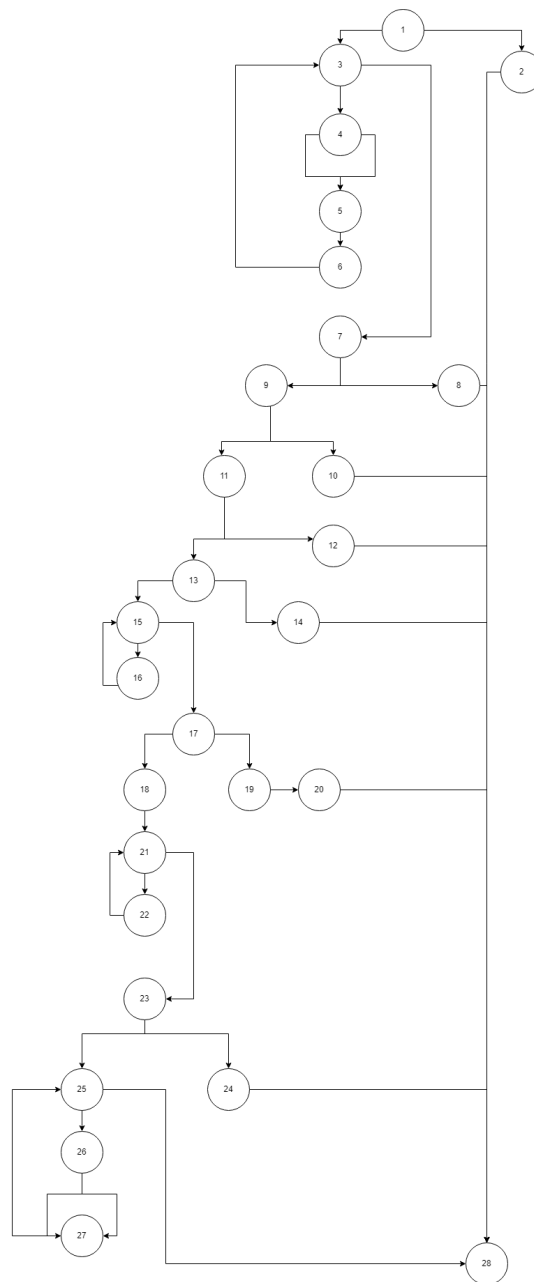
    i++;
    /*21*/ while(i<promociones.size() && !promociones.get(i).isActiva() && producto.getId()!=promociones.get(i).getProduct
        i++; //22
    if(!(i<promociones.size())) //23
        /*24*/ throw new DatosIncorrectosException("No hay suficientes promociones activas. Se necesitan al menos dos p

    ArrayList<Mesa> mesas= Sistema.getInstance().getMesas();
    for(Mesa mesa: mesas) { //25
        if(comanda.getMesa().getId()== mesa.getId()) //26
            mesa.setEstado(EstadosMesa.OCUPADA.getEstado()); //26.1
    } //27

    Sistema.getInstance().getComandas().add(comanda);

} //28

```



Verificaré la complejidad ciclomática con las tres fórmulas para evitar errores ya que es un grafo grande.

Complejidad ciclomática=

$$\text{aristas} - \text{nodos} + 2 = 42 - 30 + 2 = \mathbf{14}$$

$$\text{Nodos condición} + 1 = 13 + 1 = \mathbf{14}$$

$$\text{Áreas encerradas} + 1 = 13 + 1 = \mathbf{14}$$

Por ende concluimos que habrá como máximo **14** caminos para realizar una cobertura de decisión.

Para determinarlos, utilizamos el método simplificado:

Camino 1: 1-2-28

Camino 2: 1-3-7-8-28

Camino 3: 1-3-4-5-6-3-7-8-28

Camino 4: 1-3-4-4.1-5-6-3-7-9-10-28

Camino 5: 1-3-4-4.1-5-6-3-7-9-11-12-28

Camino 6: 1-3-4-4.1-5-6-3-7-9-11-13-14-28

Camino 7: 1-3-4-4.1-5-6-3-7-9-11-13-15-17-19-20-28

Camino 8: 1-3-4-4.1-5-6-3-7-9-11-13-15-16-15-17-18-21-23-24-28

Camino 9: 1-3-4-4.1-5-6-3-7-9-11-13-15-16-15-17-18-21-22-23-25-26-27-25-28

Camino 10: 1-3-4-4.1-5-6-3-7-9-11-13-15-16-15-17-18-21-22-23-25-26-26.1-27-25-28

Recortamos los 4 caminos restantes ya que representan caminos imposibles para nuestro código.

| Camino | Caso | Salida esperada |
|--------|--|---------------------------|
| 1 | Comanda nula | DatosIncorrectosException |
| 2 | No hay mesas | DatosIncorrectosException |
| 3 | No hay mesas libres | DatosIncorrectosException |
| 4 | La mesa de la comanda está ocupada | DatosIncorrectosException |
| 5 | El mozo asociado a la comanda es nulo o no está disponible | DatosIncorrectosException |
| 6 | La colección de promociones es nula | DatosIncorrectosException |
| 7 | La colección de promociones está vacía | DatosIncorrectosException |
| 8 | Hay solo un producto en promoción y no es el primero | DatosIncorrectosException |
| 9 | La mesa de la comanda no existe en la colección de mesas | - |
| 10 | La mesa de la comanda existe en la colección de mesas | Se da de alta la comanda |

Una vez ejecutados los junit de caja blanca y realizando la cobertura...

altaMesa

```
*/
public void altaMesa(int cantComensales, String estado) throws DatosIncorrectosException {
    if(estado==null) //1
        throw new DatosIncorrectosException("Estado nulo."); //2
    if(Sistema.getInstance().getMesas().size() >= 1) { //3
        if(cantComensales >= 2) { //4
            Sistema.getInstance().getMesas().add(new Mesa(cantComensales, estado));
        }
        else //5
            throw new DatosIncorrectosException("Cantidad de comensales debe ser mayor a 1.");
    }
    else //6
        Sistema.getInstance().getMesas().add(new Mesa(cantComensales, estado));
} //7
```

modificarMesa

```
public void modificarMesa(Mesa mesa, int cantComensales, String estado) throws DatosIncorrectosException {
    if( mesa==null || estado==null) //1
        throw new DatosIncorrectosException("Parametros invalidos.");
    if(mesa.getId() >= 1) { //2
        if(cantComensales >= 2) { //3
            int i = 0;
            ArrayList<Mesa> mesas = Sistema.getInstance().getMesas();
            while(i < mesas.size() && mesas.get(i).getId() != mesa.getId()) //4
                i++; //5
            mesas.get(i).setCantComensales(cantComensales);
            mesas.get(i).setEstado(estado);
        } //6
        else //7
            throw new DatosIncorrectosException("Cantidad de comensales debe ser mayor a 1.");
    } //8
    else { //9
        int i = 0;
        ArrayList<Mesa> mesas = Sistema.getInstance().getMesas();
        while(i < mesas.size() && mesas.get(i).getId() != mesa.getId()) //10
            i++; //11
        mesas.get(i).setCantComensales(cantComensales);
        mesas.get(i).setEstado(estado);
    } //12
} //13
```


altaComanda:

```
60 public void altaComanda(Comanda comanda) throws DatosIncorrectosException {
61     Mozo mozoAsociado;
62     int i=0;
63     boolean haymesaLibre=false;
64     Producto producto;
65     if(comanda==null) //1
66         throw new DatosIncorrectosException("Parametros invalidos."); //2
67     for(Mesa mesa: Sistema.getInstance().getMesas()) { //3
68         if(mesa.getEstado().equals(EstadosMesa.LIBRE.getEstado())){ //4
69             haymesaLibre=true; //4.1
70         } //5
71     } //6
72     if(!haymesaLibre) //7
73         throw new DatosIncorrectosException("No hay mesas libres en el restaurante"); //8
74
75     if(comanda.getMesa().getEstado().equals(EstadosMesa.OCUPADA.getEstado()) ) //9
76         throw new DatosIncorrectosException("La mesa esta ocupada, debe elegir una mesa activa."); //10
77
78     mozoAsociado =Sistema.getInstance().getAsignacionesDesdeMesa().get(comanda.getMesa().getId()).getMozo();
79
80     if( mozoAsociado==null || !mozoAsociado.getEstado().equals(Estado.ACTIVO.getEstado())) //11
81         throw new DatosIncorrectosException("El mozo asociado no existe o no se encuentra disponible."); //12
82
83     /*De aca para abajo controla que haya al menos dos promociones activas*/
84     ArrayList<Promocion> promociones=Sistema.getInstance().getPromociones();
85     if(promociones==null) //13
86         throw new DatosIncorrectosException("No hay promociones activas."); //14
87
88     while(i<promociones.size() && !promociones.get(i).isActiva()) { //15 la condicion aqui estaba mal
89         i++; //16
90     }
91     if(i<promociones.size()) //17
92         producto=promociones.get(i).getProducto(); //18
93
94     else //19
95         throw new DatosIncorrectosException("No hay promociones activas."); //20
96
97     i++;
98     while(i<promociones.size() && !promociones.get(i).isActiva() && producto.getId()!=promociones.get(i).getProducto().getId()) //21
99         i++; //22
100     if(!i<promociones.size()) //23
101         throw new DatosIncorrectosException("No hay suficientes promociones activas. Se necesitan al menos dos promociones de dos productos distintos");//24
102
103     ArrayList<Mesa> mesas= Sistema.getInstance().getMesas();
104     for(Mesa mesa: mesas) { //25
105         if(comanda.getMesa().getId()== mesa.getId()) //26
106             mesa.setEstado(EstadosMesa.OCUPADA.getEstado()); //26.1
107     } //27
108
109     Sistema.getInstance().getComandas().add(comanda);
110
111
112
113 } //28
```

Test de Persistencia

Durante las pruebas se testean los siguientes escenarios:

- La creación correcta del archivo.

- La escritura en el archivo.
- 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

Para la realización del test de GUI se utilizó la clase Robot y se testearon las ventanas “VistaLogin”, “VistaMenuOperarioAdministrador” y “VistaFacturacionYPedidos”.

Las componentes de las ventanas originalmente no tenían nombre, por lo que fueron agregados para no tener problemas con el método `getComponentForName(nombre)`.

Por suerte no hubo que testear ventanas modales (como los `JOptionPane`), ya que al haber este tipo de ventanas se complicaría mucho más las cosas porque las componentes de estas ventanas no tienen nombre.

Todos estos test corren con JUnit 4 porque no corren correctamente con JUnit 5.

VistaLogin

En la VistaLogin se testearon los casos cuando se loguea un administrador, un operario no administrador, cuando los datos ingresados son incorrectos, cuando uno de los campos está vacío y con ambos campos vacíos.

Cuando es el primer ingreso, al rellenar el nombre de usuario con “ADMIN” y con la contraseña “ADMIN1234”, lo ideal es que el sistema nos deje ingresar y cambiar la contraseña. Sin embargo, aparece un mensaje diciendo “Usuario incorrecto”, lo cual es un error de no cumplir con la SRS.

El caso de usuario inexistente es el mismo que usuario incorrecto y aparece un mensaje diciendo “Usuario incorrecto”, que es lo esperado. También ocurre lo mismo cuando el campo del nombre de usuario está vacío.

En los casos de contraseña incorrecta y el campo de contraseña vacío aparece un mensaje diciendo “*Password* incorrecta”, que es lo esperado.

En el caso de ambos campos vacíos aparece un mensaje diciendo “Usuario incorrecto”, que es lo esperado, ya que es la primera excepción que salta.

Cuando se rellena el nombre de usuario con “G” y con la contraseña “H”, si está activa la persistencia, el sistema nos debería dejar entrar como administrador que es lo esperado. Caso contrario se obtiene el mismo caso de usuario inexistente. Algo parecido ocurre cuando se rellena el nombre de usuario con “A” y con la contraseña “Imanol1”, si está activa la persistencia, el sistema nos debería dejar entrar como un operario que también es lo esperado. Pero si un usuario está inactivo, si bien no nos deja ingresar al sistema, debería encontrarse con un mensaje diciendo “Usuario inactivo”, pero lo encontrado fue “*Password* incorrecta”.

VistaMenuOperarioAdministrador

En esta vista se testeó a ver qué pasaba al hacer clic en cada botón de la interfaz gráfica. Para que el programa pueda ejecutarse correctamente se comentó parte del código del método main de la clase App para evitar que los mozos aparezcan más de una vez.

Al hacer clic en el botón “Dar inicio al día”, dependiendo de la cantidad de mozos activos y de Franco, se habilitará el botón “Facturación y pedidos” o aparecerán distintos mensajes de error.

1. Para el caso con 4 mozos activos y 2 de Franco: se habilitará el botón “Facturación y pedidos”.
2. Para el caso sin mozos (ya sean activo o de Franco): aparecerá el mensaje “Se necesita al menos 1 mozo en estado activo para iniciar el día”. Por SRS deberían haber 4 activos.
3. Para el caso con todos los mozos activos: aparecerá el mensaje “Se necesitan 2 mozos de franco para iniciar el día.”.
4. Para el caso con un mozo activo y el resto de Franco: aparecerá el mensaje “Se necesitan 2 mozos de franco para iniciar el día.”.
5. Para el caso con todos los mozos de Franco: aparecerá el mensaje “Se necesita al menos 1 mozo en estado activo para iniciar el día”.

Se han probado todos los casos y debido a la persistencia, sólo funcionó correctamente el caso 1.

Para el resto de los botones lo que se esperaba es que cambie de ventana de acuerdo al botón clickeado y es lo que ocurrió.

VistaFacturacionYPedidos

En esta vista se testeó a ver qué pasaba al hacer clic en cada componente.

Cuando se hace clic en el botón “Atrás”, lo que pasaba era que, aunque estuvieras logueado como operario no admin, te mandaba al menú del admin. Eso se corrigió desde el controlador, y dependiendo del tipo de usuario, te manda al menú del admin o al login en caso de ser un operario no admin.

Lo que hace el robot al hacer clic en el comboBox es bajar el cursor del mouse y elegir del menú desplegable una comanda.

Depende de si la lista de comandas es null o está vacía, los botones “Agregar pedido” y “Cerrar comanda” estarán deshabilitados, en caso contrario, estarán habilitados. Esto fue arreglado porque anteriormente si la lista de comandas es null crasheaba e iban a estar habilitados ambos botones.

Lo que se esperaba es que al hacer clic en los botones “Agregar pedido” y “Cerrar comanda” también cambie de ventana de acuerdo al botón clickeado y es lo que ocurrió.

Test de Integración

Caso de uso “Login”:

Descripción: El usuario (ya sea el administrador o un operario) debe identificarse con un nombre de usuario y una contraseña para poder acceder al sistema.

Actores: Usuario.

Pre condiciones:

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

Flujo Normal:

1. El usuario accede al sistema.
2. El sistema solicita credenciales.
3. El usuario ingresa su nombre de usuario y su contraseña.
4. El sistema valida las credenciales del usuario e ingresará al sistema.

Flujo Alternativo 1:

1. El usuario administrador ingresa por primera vez al sistema.

Flujo Alternativo 2:

1. No hay operarios en la colección.

Flujo Alternativo 3:

1. El usuario no existe (o es incorrecto).

Flujo Alternativo 4:

1. La contraseña es incorrecta.

Flujo Alternativo 5:

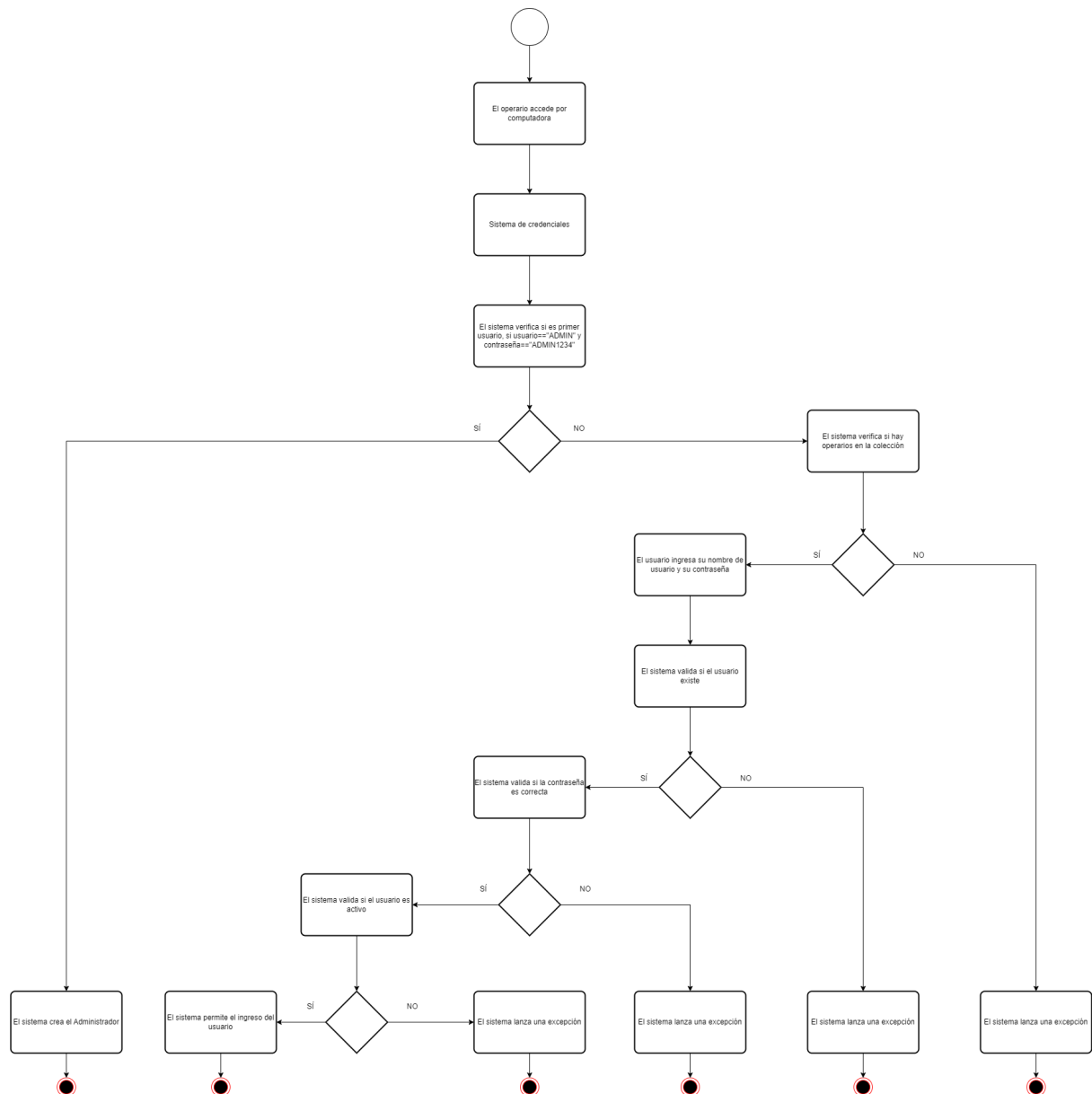
1. El estado del usuario es inactivo.

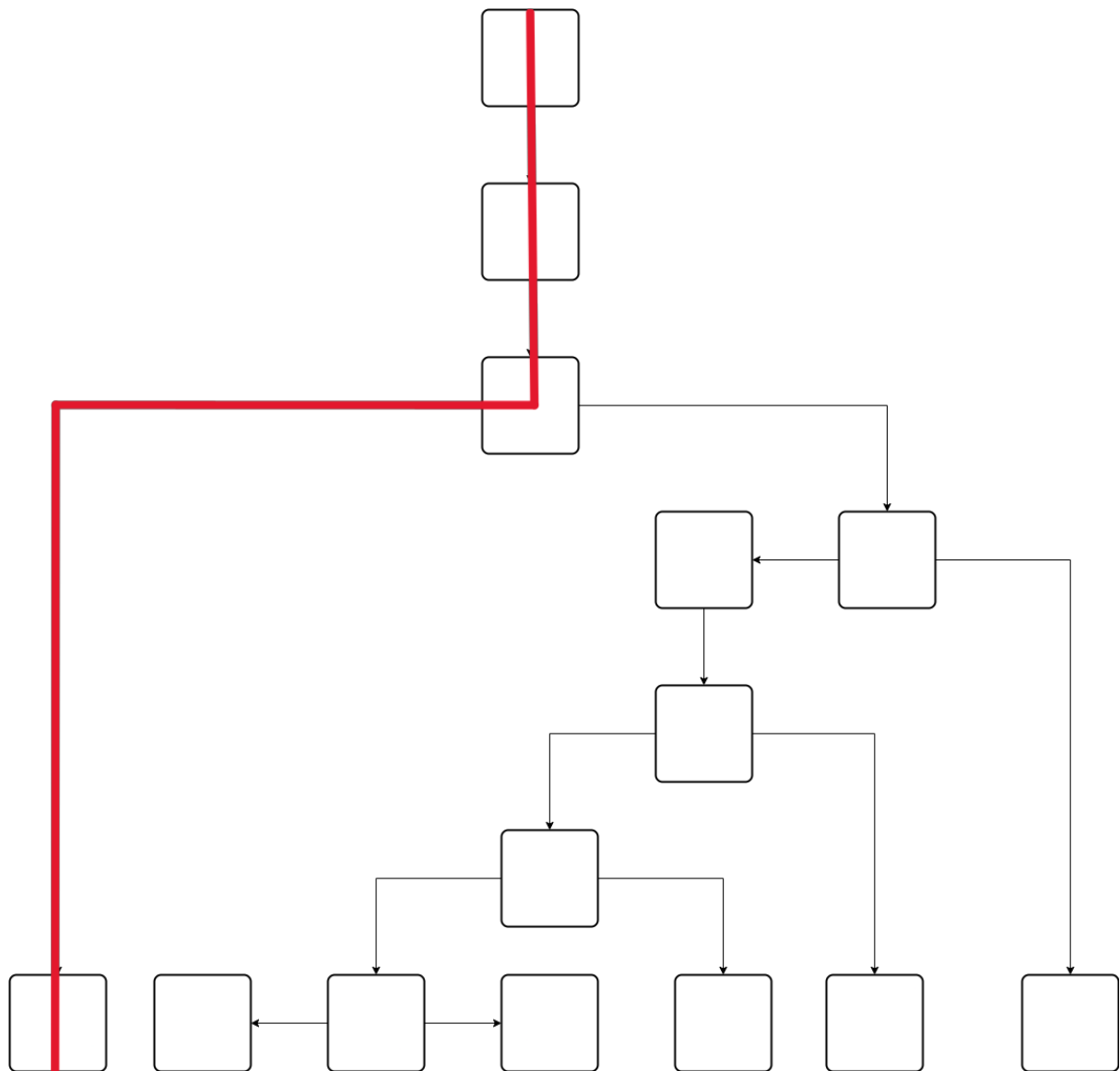
Excepciones:

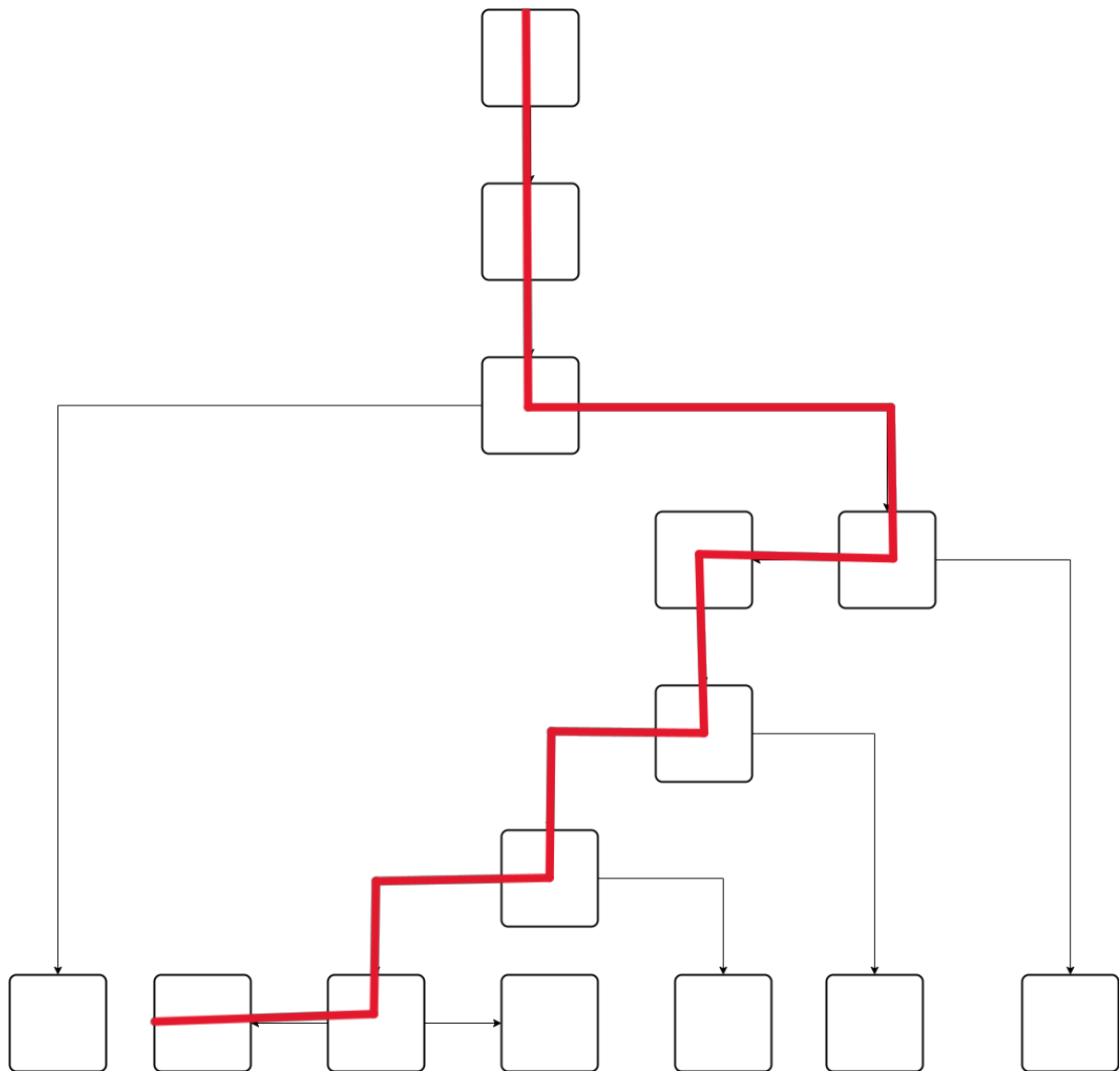
1. E1. WrongUserException.
2. E2. WrongPasswordException.

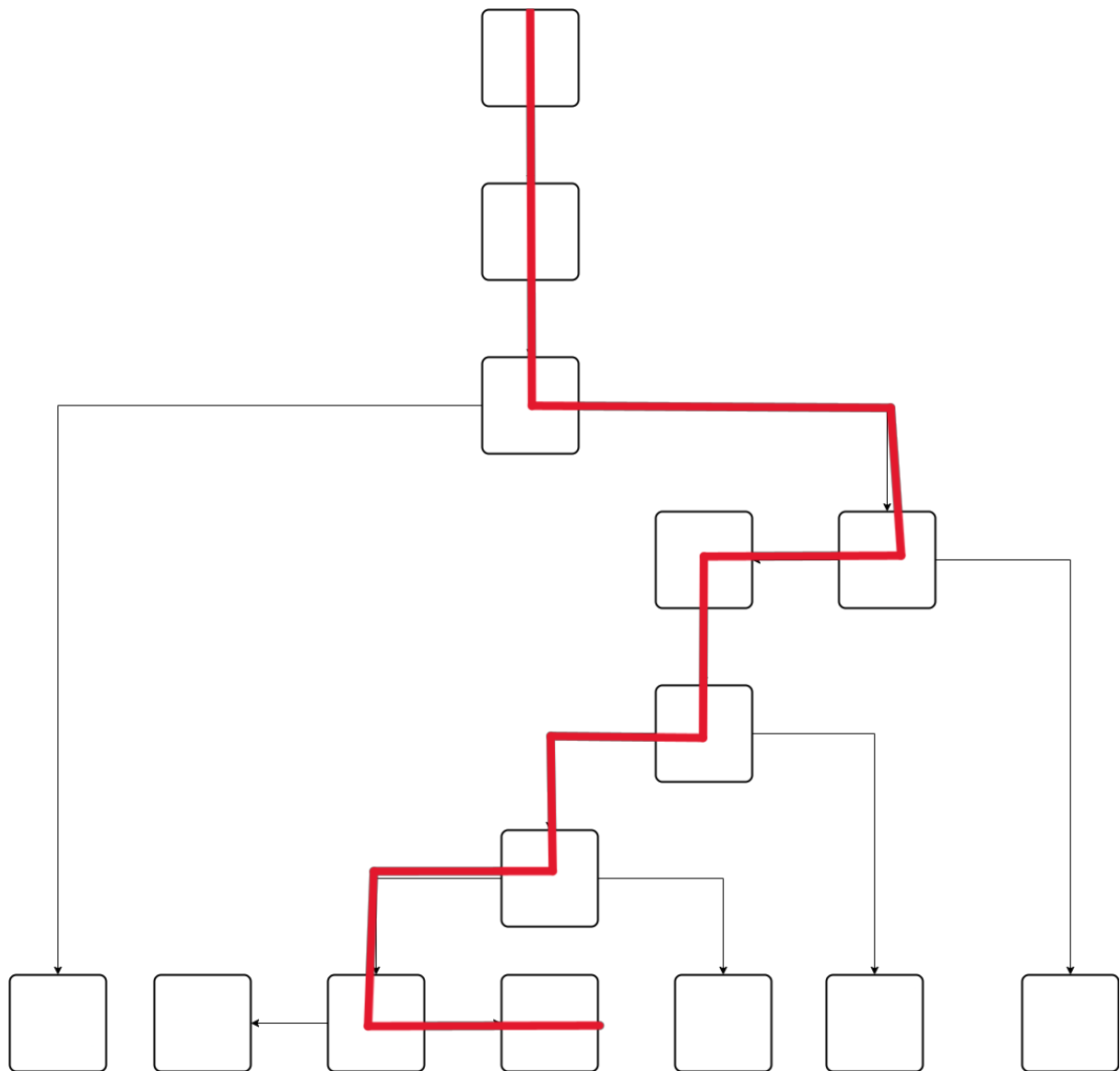
Post condiciones:

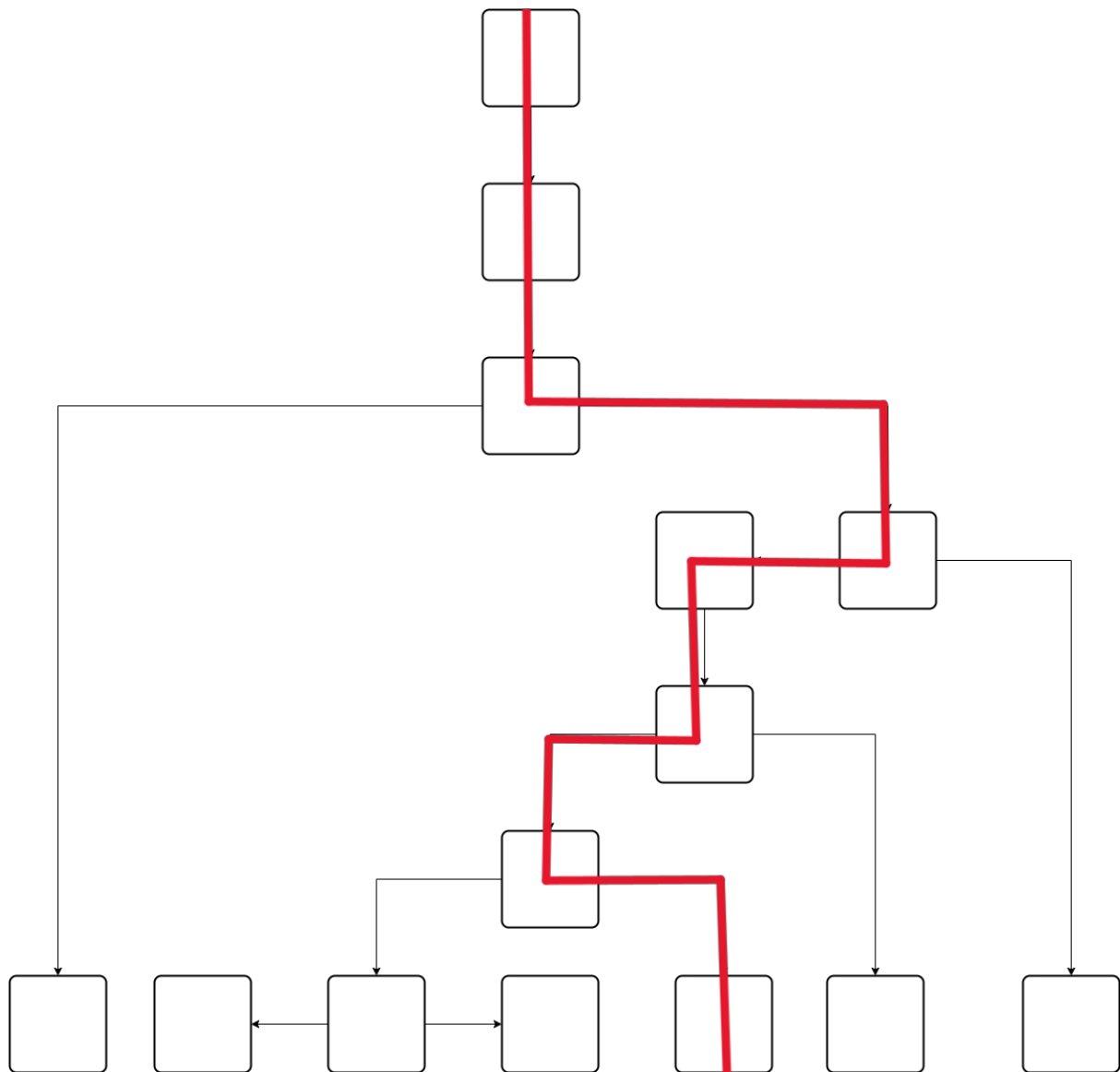
1. El usuario accede al sistema.

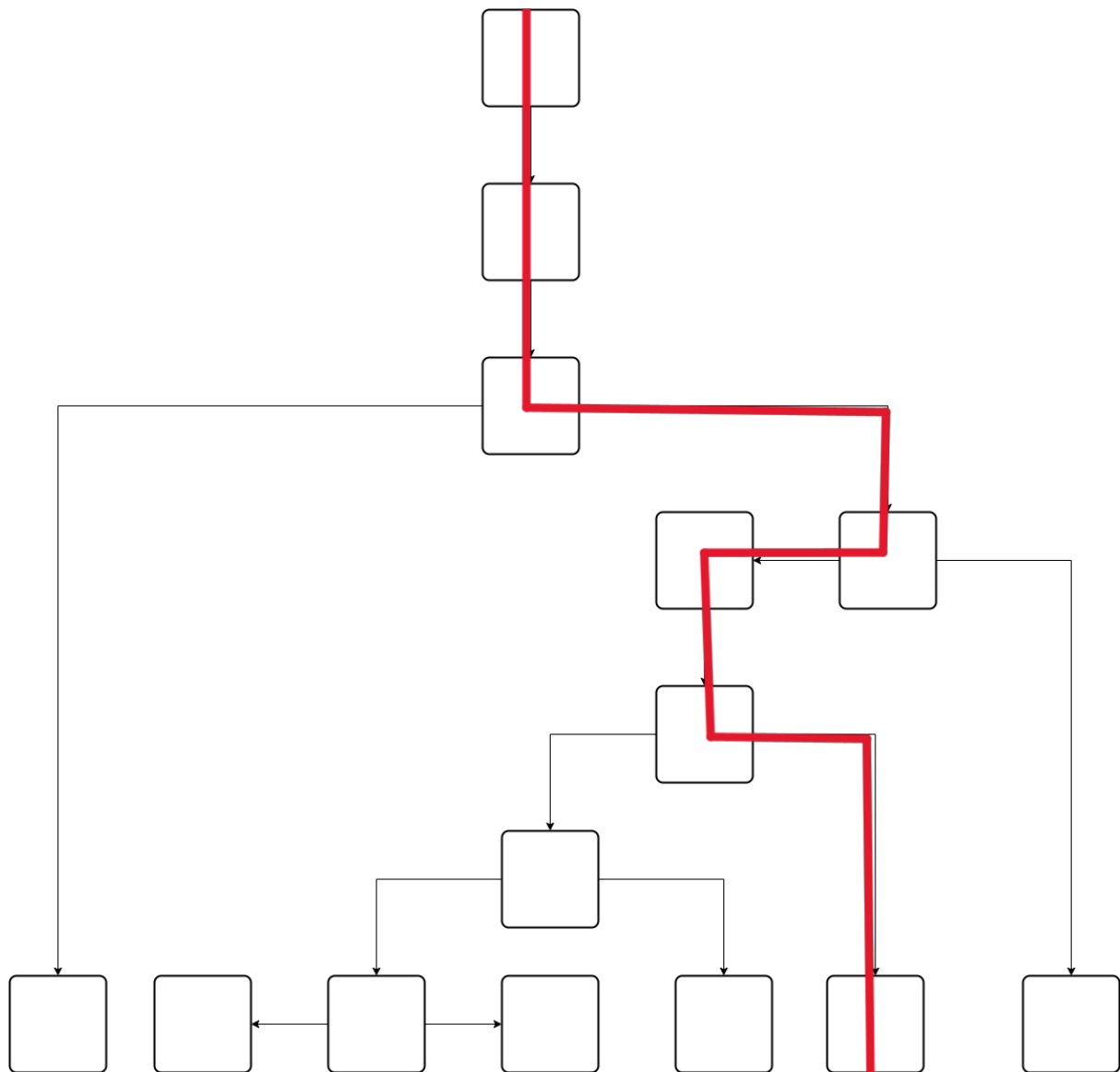


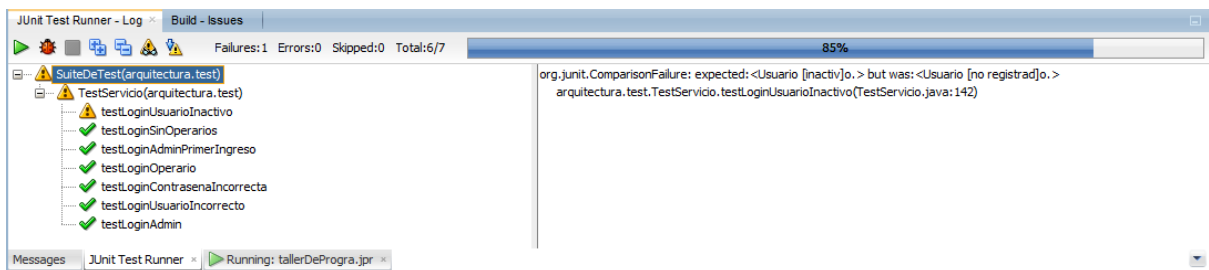
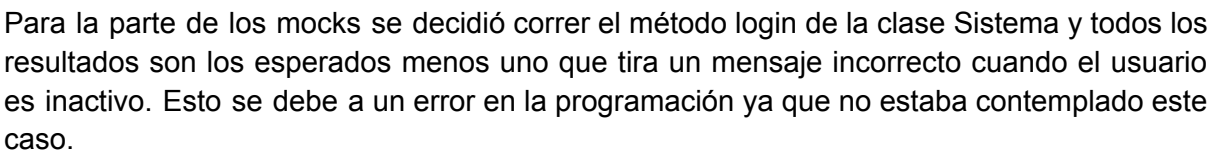












Conclusión

A lo largo de este trabajo pudimos implementar diversos métodos de testing. Detectamos que por momentos la SRS presentaba algunos fallos de especificación y análisis, como por ejemplo los casos analizados en las pruebas de caja negra.

Al momento de hacer dichas pruebas, priorizamos a los métodos estructurales del 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 a realizar las pruebas de caja negra para determinar casos de testeo y los resultados esperados.

Utilizamos las pruebas de caja blanca para verificar el funcionamiento de las líneas de código que no se llegaron a ejecutar luego de realizado el test de cobertura. Para eso realizamos el gráfico ciclomático de los métodos escogidos y luego calculamos la complejidad de los mismos, y concluimos haciendo los caminos básicos con su correspondiente salida esperada. Una vez hecho esto, armamos los JUnit test donde se correría cada uno de los posibles caminos y pudiendo testear los métodos.

Durante la ejecución de pruebas de persistencia no se encontraron errores, por lo que concluimos que su funcionamiento es correcto.

Durante la ejecución del test de GUI, los test funcionaban dependiendo de lo persistido, ya que funciona muy distinto si se carga con la persistencia a cuando se ingresa al sistema por primera vez.

Por último, en la ejecución de los test de integración, el único problema fue un caso que de arreglarse funcionaría correctamente.