Trabajo Práctico Especial

Taller de Programación I

Testers:
Aureliano Vega Imbalde
Paula Bonifazi Aquino

Desarrolladores:
Martiniano Presa
Valentina Reale

Vídeo: https://www.youtube.com/watch?v=L8fMaxvc5p4

Introducción

En este trabajo práctico se llevó a cabo el desarrollo de un sistema informático con el objetivo de administrar el funcionamiento diario de un local gastronómico.
Para dicha realización, la cátedra proporcionó el documento de Especificación Requerimiento de Software. Dicho documento es de suma importancia ya que permite definir las pre y post condiciones tanto a los desarrolladores como a los testers.
A partir de las Especificaciones se realizó el test Unitario de Caja Negra, contemplando la documentación de cada clase y las pre y postcondiciones de cada método, con los escenarios y baterías de pruebas para uno. Luego se prosiguió con el Test de Cobertura, o Caja Blanca, para poder cubrir los casos que no fueron contemplados en el test anterior. Se finalizó con Test de integración, optando por un método representativo que abarca gran parte de los métodos previamente testeados para analizar el comportamiento de las diferentes clases en juego.
Por otro lado, se realizaron los Test de Persistencia de datos para poder verificar errores en la lectura o escritura de archivos y Test de interfaces gráficas utilizando la clase Robot para una ventana elegida y así poder verificar si el ingreso de datos fue correcto.

Caja negra

En esta parte del trabajo práctico se llevó a cabo tests de caja negra. En el código se testeó todos los métodos con sus atributos en todas las variantes posibles (siempre respetando el contrato). Para que el informe no se extienda demasiado, se presenta la tabla de particiones y la batería de pruebas para un constructor y un método significativo. Luego de eso, se presentan detalladamente todas las fallas y errores encontrados al aplicar testeo en caja negra.

Demostración

Como se mencionó, se realizará la tabla de particiones y la batería de pruebas del constructor de la clase Operario, su setter de nyA y del método agregaMozo de la clase FuncionalidadAdmin a modo de demostración:

- Clase Operario:

Operario(String nyA,String userName,String password,boolean activo)

Tabla de particiones

Condiciones de entrada	Clases válidas	Clases inválidas
nyA	alfabético no vacío (1)	vacío (2)
		null (3)

userName	alfabético no vacío (4)	vacío (5)
		null (6)
password	alfabético no vacío (7)	vacío (8)
		null (9)
activo	true (10)	
	false (11)	

Batería de pruebas

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Salida esperada	Salida obtenida
correcta	{“Roberto Rodriguez”, “RobertoR”, “Roberto123”, true}	1,4, 7, 10	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro
correcta	{“Roberto Rodriguez”, “RobertoR”, “Roberto123”, false}	11	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro
incorrecta	{null, “RobertoR”, “Roberto123”, false}	3	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro
incorrecta	{Roberto Rodriguez, null, Roberto123, false}	6	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro
incorrecta	{Roberto Rodriguez, RobertoR, null, false}	9	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro
incorrecta	{“”, RobertoR, Roberto123, false}	2	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro

incorrecta	{Roberto Rodriguez, “”, Roberto123, false}	5	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro
incorrecta	{Roberto Rodriguez, RobertoR, “”, false}	8	Se crea un nuevo operario con los valores pasados como parámetro	Se crea un nuevo operario con los valores pasados como parámetro

void setNyA(String nyA)

Tabla de particiones

Condiciones de entrada	Clases válidas	Clases inválidas
NyA	alfabético no vacío (1)	vacío (2)
		null (3)

Batería de pruebas

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Salida esperada	Salida obtenida
correcta	{“Akini Jing”}	1	el atributo NyA de operario se modifica a “Akini Jing”	el atributo NyA de operario se modifica a “Akini Jing”
incorrecta	{“”}	2	el atributo NyA de operario se modifica a “”	el atributo NyA de operario se modifica a “”
incorrecta	{null}	3	el atributo NyA de operario se modifica a null	el atributo NyA de operario se modifica a null

- Clase FuncionaliadAdmin:

void agregaMozo(String NyA, GregorianCalendar fechaNacimiento, int cantHijos, Enumerados.estadoMozo estado) throws EdadInvalida_Exception, CantHijosInvalida_Exception, NyARepetido_Exception

Escenarios

Número de escenario	Descripción
1	con datos en el sistema
2	con un mozo con NyA = “Paula” en el Sistema
3	con el sistema vacío

Tabla de particiones

Condiciones de entrada	Clases válidas	Clases inválidas
NyA	alfabético no vacío no repetido (1)	repetido (2)
fechaNacimiento	mayor a 18 años (3)	menor a 18 años (4)
		null (11)
cantHijos	positivo o 0 (5)	negativo (6)
estado	Activo (7)	null (10)
	Ausente (8)	
	De franco (9)	

Aclaración: el contrato nunca menciona la posibilidad o imposibilidad de pasar un valor igual a null a las variables estado y fechaNacimiento, por lo tanto son valores a testear.

Batería de pruebas

Tanto para el escenario 1 como para el escenario 3

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Salida esperada	Salida obtenida
correcta	{"Aureliano Vega Imbalde", new GregorianCalendar (2000, 11, 15), 2, Enumerados.estadoMozo.ACTIVO}	1, 3, 5, 7	Se agrega un nuevo Mozo al Sistema con los valores pasados como parámetro.	Se agrega un nuevo Mozo al Sistema con los valores pasados como parámetro.

correcta	{"Aureliano Vega Imbalde", new GregorianCalendar (2000, 11, 15), 2, Enumerados.estadoMozo.AUSENTE}	8	Se agrega un nuevo Mozo al Sistema con los valores pasados como parámetro.	mozo.getEstado( )= ACTIVO *Failure*
correcta	{"Aureliano Vega Imbalde", new GregorianCalendar (2000, 11, 15), 2, Enumerados.estadoMozo.DEFRANCO}	9	Se agrega un nuevo Mozo al Sistema con los valores pasados como parámetro.	mozo.getEstado( )= ACTIVO *Failure*
incorrecta	{"Aureliano Vega Imbalde", new GregorianCalendar (2000, 11, 15), 2, null}	10	Se agrega un nuevo Mozo al Sistema con los valores pasados como parámetro.	mozo.getEstado( )= ACTIVO *Failure*
incorrecta	{"Aureliano Vega Imbalde", new GregorianCalendar (2010, 11, 15), 2, Enumerados.estadoMozo.ACTIVO}	4	EdadInvalida_Exception	EdadInvalida_Exception
incorrecta	{"Aureliano Vega Imbalde", null, 2, Enumerados.estadoMozo.ACTIVO}	11	Se agrega un nuevo Mozo al Sistema con los valores pasados como parámetro.	*Error*
incorrecta	"Aureliano Vega Imbalde", new GregorianCalendar (2000, 11, 15), -2, Enumerados.estadoMozo.ACTIVO	6	CantHijosInvalida_Exception	CantHijosInvalida_Exception

Para el escenario 2:

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Salida esperada	Salida obtenida
incorrecta	{"Paula", new GregorianCalendar (2000, 11, 15), 2, Enumerados.estadoMozo.ACTIVO}	2	NyARepetido_Exception	NyARepetido_Exception

Errores y fallas encontrados al testear

- Clase Administrador:

El constructor de Administrador es vacío, por lo que no se puede testear. El testeo de los setters se encuentra en la sección de Operario ya que administrador se extiende de operario.

- Clase Comanda:

void Comanda(Mesa mesa, estadoComanda estado)

Valores de entrada	Salida esperada	Salida obtenida
{null,estadoComanda.CERRADO}	Se crea un nueva comanda con los valores pasados como parámetro	*Error*

Al crear una nueva comanda con una mesa = null salta error. La documentación no hace mención a la posibilidad de que Mesa sea null.

void agregaPedido(Pedido pedido)

Este método no se pudo testear porque no tiene ni descripción ni postcondiciones indicadas. Entonces, al no saber lo que hace, no se sabe qué hay que testear. Además no habría forma de saber qué poner en la comparación de Assert.assertEquals()

- Clase MesaAtendida:

void agregaPromocion(PromocionProd promocion)

Este método no se pudo testear porque no tiene ni descripción ni postcondiciones indicadas. Entonces, al no saber lo que hace, no se sabe qué hay que testear. Además no habría forma de saber qué poner en la comparación de Assert.assertEquals()

- Clase Mozo:

Mozo(String NyA, int cantHijos)

Una de las precondiciones indica que el mozo debe ser mayor a 18 años, sin embargo, no hay ninguna fecha que se pase como parámetro.

- Clase Operario:

void modificaOperario(String NyA, String userName, String password, boolean activo)

Este método no se pudo testear porque no tiene ni descripción ni postcondiciones indicadas. Entonces, al no saber lo que hace, no se sabe qué hay que testear. Además no habría forma de saber qué poner en la comparación de Assert.assertEquals()

boolean verificaPassword(String password)

Este método no se pudo testear porque no tiene ni descripción ni postcondiciones indicadas. Entonces, al no saber lo que hace, no se sabe qué hay que testear. Además no habría forma de saber qué poner en la comparación de Assert.assertEquals()

- Clase Sueldo:

static void setRemBasic(double remBasic)

Valores de entrada	Salida esperada	Salida obtenida
{279}	Sueldo.getRemBasic() = 279	Sueldo.getRemBasic() = 5000 *Failure*

void calculaSueldo(int cantHijos)

(Setteando remBasic a 279)

Valores de entrada	Salida esperada	Salida obtenida
{3}	320.85	5750 *Failure*

- Clase FuncionalidadAdmin:

void registraOperario(String NyA, String userName, String password, Enumerados.estadoOperario estado) throws UserNameRepetido_Exception, ContrasenaIncorrecta_Exception

Habiendo un usuario con el username “aureeliano” en la colección previamente:

Valores de entrada	Salida esperada	Salida obtenida
{"Marianela", “aureeliano”, "cataratas123",Enumerados.estadoOperario.ACTIVO}	UserNameRepetido_Exception	ContrasenaIncorrecta_Exception *Failure*
{"Carola", "caritenss.gl", "perfumew0rlddominati1on", Enumerados.estadoOperario.ACTIVO}	Se agrega un nuevo Operario al sistema con los atributos pasados como parámetro	ContrasenaIncorrecta_Exception *Failure*
{"Carola", "caritenss.gl", "perfumew0rlddominati1on", Enumerados.estadoOperario.INACTIVO}	Se agrega un nuevo Operario al sistema con los atributos pasados como parámetro	ContrasenaIncorrecta_Exception *Failure*
{"Carola", "caritenss.gl", "perfumew0rlddominati1on", null}	Se agrega un nuevo Operario al sistema con los atributos pasados como parámetro	ContrasenaIncorrecta_Exception *Failure*

El contrato no indica cuándo se lanza ContrasenaIncorrecta_Exception. No hay forma de que el tester sepa qué valor introducir en password para que no se lance esa excepción.

void agregaProducto(String nombre, double precioCosto, double precioVenta, int stockInicial) throws precioInvalido_Exception

Valores de entrada	Salida esperada	Salida obtenida
{"Chorizo Cake",4000,20,3}	Se agrega un nuevo Producto al sistema con los atributos pasados como parámetro	precioInvalido_Exception *Failure*

El contrato indica que la excepción precioInvalido_Exception se lanza cuando alguno de los precios es negativo. Sin embargo, no menciona nada sobre un precio de costo mayor al precio de venta, por lo que no debería lanzarse la excepción.

void modificaRemuneracionBasica(double remBasica)

Valores de entrada	Salida esperada	Salida obtenida
{43}	sueldo.getRemBasic = 43	sueldo.getRemBasic = 5000

		Failure

- Clase FuncionalidadOperario:

void estableceEstadosMozos(Enumerados.estadoMozo estado, String nya)

Valores de entrada	Salida esperada	Salida obtenida
{Enumerados.estadoMozo.ACTIVO, nyA de mozo fuera del sistema}	se cambia el estado del mozo a ACTIVO	*Error*

El contrato no indica qué sucede si se pasa como parámetro el nombre de un mozo que no se encuentra en el sistema.

void modificaOperario(String NyA, String userName, String password) throws UserNameRepetido_Exception, ContrasenaIncorrecta_Exception

Valores de entrada	Salida esperada	Salida obtenida
{"nuevo nombre", "nuevo username", "nueva password"}	Se modifican el nyA, el username y la password del operario que inició sesión	ContrasenaIncorrecta_Exception *Failure*

El contrato no indica cuándo se lanza ContrasenaIncorrecta_Exception. No hay forma de que el tester sepa qué valor introducir en password para que no se lance esa excepción.

void modificaMozo(Mozo mozo, Enumerados.estadoMozo estado, int cantHijos) throws CantHijosInvalida_Exception, NyARepetido_Exception

Valores de entrada	Salida esperada	Salida obtenida
{null, Enumerados.estadoMozo.AUSENTE, 19}	Se modifican el estado y cantHijos del mozo pasado como parámetro	*Error*

El contrato no indica qué sucede si se pasa como parámetro un mozo null.

Por otro lado, según el contrato, NyARepetido_Exception se lanza cuando se quiere modificar nyA de un mozo por uno que ya posee otro mozo. Sin embargo, al método no se pasa como parámetro ningún valor String.

void modificaProducto(int idProd, String nombre, double precioCosto, double precioVenta, int stockInicial throws NoExisteID_Exception, precioInvalido_Exception, prodEnUso_Exception

Valores de entrada	Salida esperada	Salida obtenida
{1,"ninios envolvidos", -1900, 2000, 3}	Se modifican los atributos del producto	precioInvalido_Exception *Failure*
{1,"ninios envolvidos", 1900, -2000, 3}	Se modifican los atributos del producto	precioInvalido_Exception *Failure*

El contrato no indica cuándo se lanzan precioInvalido_Exception y prodEnUso_Exception.

void modificaMesa(int nroMesa, int cantPax, Enumerados.estadoMesa estado) throws NoExisteMesa_Exception, CantComensalesInvalida_Exception

Valores de entrada	Salida esperada	Salida obtenida
{0, 1, Enumerados.estadoMesa. LIBRE}	CantComensalesInvalida_Exception	Se modifican los atributos de la mesa *Failure*

Según el contrato, CantComensalesInvalida_Exception se lanza cuando cantPax (cantidad de comensales) es menor que 2.

void agregaPromocionProd(int idProd, Enumerados.diasDePromo dia, boolean aplica2x1, boolean aplicaDtoPorCantidad, int dtoPorCantidad_CantMinima, double dtoPorCantidad_PrecioUnitario, boolean activa) throws PromoInvalida_Exception, NoExisteID_Exception

Valores de entrada	Salida esperada	Salida obtenida
{9999,Enumerados.diasDePromo.LUNES,true,, true, 3, 0.5, true**}**	Se agrega una nueva PromocionProd al Sistema.	NoExisteID_Exception *Failure*

Según la documentación, PromoIdRepetido_Exception se lanza al intentar asignar un identificador perteneciente a otra promocion. Sin embargo, en la cabecera del método se puede observar que esa excepción nunca se lanza. Por otro lado, la documentación no especifica qué hace NoExisteID_Exception. Por último, la excepción PromoInvalida_Exception está en la cabecera del método pero en la documentación no.

void modificaPromocionProd(int idProm, boolean activa, Enumerados.diasDePromo dia, boolean aplica2x1, boolean aplicaDtoPorCantidad, int dtoPorCantidad_CantMinima, double dtoPorCantidad_PrecioUnitario) throws PromoInvalida_Exception, NoExisteID_Exception

Valores de entrada	Salida esperada	Salida obtenida
{id de promocionProd que no se encuentra en el sistema, true, Enumerados.diasDePromo.LUNES, true, true, 3, 0.2**}**	PromocionProd con los atributos modificados	NoExisteID_Exception *Failure*
{0, true, Enumerados.diasDePromo.LUNES, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*
{0, true, Enumerados.diasDePromo.MARTES, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*
{0, true, Enumerados.diasDePromo.MIERCOLES, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*
{0, true, Enumerados.diasDePromo	PromocionProd con los atributos	diasDePromo de promocionProd no se modifica

.JUEVES, true, true, 3, 0.2}	modificados	*Failure*
{0, true, Enumerados.diasDePromo .VIERNES, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*
{0, true, Enumerados.diasDePromo .SABADO, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*
{0, true, Enumerados.diasDePromo .DOMINGO, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*
{0, true, null, true, true, 3, 0.2}	PromocionProd con los atributos modificados	diasDePromo de promocionProd no se modifica *Failure*

La documentación no menciona a PromoInvalida_Exception y NoExisteID_Exception a pesar de estar en la cabecera del método. No hay forma de testearlas con caja negra. Tampoco hace referencia a la posibilidad o imposibilidad de pasar null como parámetro a la variable dia.

void eliminaPromocionProd(int idProm) throws PromoInvalida_Exception

Valores de entrada	Salida esperada	Salida obtenida
{id de promocionProd que no se encuentra en el sistema, true, Enumerados.diasDePromo .LUNES, true, true, 3, 0.2**}**	promocionProd eliminada	PromoInvalida_Exception *Failure*

La documentación no menciona a PromoInvalida_Exception a pesar de estar en la cabecera del método. No hay forma de testearla con caja negra.

void agregaPromocionTemporal(boolean activa, modelo.Enumerados.diasDePromo diasDePromo, String nombre, modelo.Enumerados.formaDePago formaDePago, int porcentajeDesc,

boolean esAcumulable, int horaInicio, int horaFinal) throws PromoRepetida_Exception

Siendo promT una promocionTemporal ya existente en el sistema:

Valores de entrada	Salida esperada	Salida obtenida
{promT.isActiva(), promT.getDiasDePromo(), promT.getNombre(), promT.getFormaDePago(), promT.getPorcentajeDesc(), promT.isEsAcumulable(), promT.getHoraInicio(), promT.getHoraFinal()}	PromoRepetida_Exception	Se agrega una nueva promocionTemporal al sistema con los atributos pasados como parámetro *Failure*

Según la documentación, PromoRepetida_Exception se lanza cuando la promocionTemporal que se desea agregar ya existe en la colección promocionesTemp del sistema.

void cierraMesa(int nroMesa, Enumerados.formaDePago formaDePago) throws MesaNoOcupadaException

Siendo mesa una Mesa con un mozo y estado = OCUPADA, y además estando asociada a una comanda:

Valores de entrada	Salida esperada	Salida obtenida
{nroMesa de mesa, Enumerados.formaDePago.CTADNI}	MesaNoOcupadaException	Cierra la mesa *Failure*

Siendo mesa una Mesa con un mozo y estado = LIBRE, y además estando asociada a una comanda:

Valores de entrada	Salida esperada	Salida obtenida
{nroMesa de mesa, Enumerados.formaDePago.CTADNI}	Estado de la comanda a la cuál está asociada la mesa = cerrada	MesaNoOcupadaException *Failure*

A pesar de que el nombre indique lo contrario, en la documentación se explica que la excepción MesaNoOcupadaException se lanza cuando la mesa es null o el estado de la mesa sigue en ocupada.

void abreComanda(Mesa mesa) throws NoExisteMesa_Exception, MesaOcupada_Exception, MozoInactivo_Exception

Valores de entrada	Salida esperada	Salida obtenida
{mesa que no se encuentra en el sistema}	Se agrega una comanda al sistema.	*Error*

Escenario: ningún mozo en estado activo.

Valores de entrada	Salida esperada	Salida obtenida
{mesa}	MozoInactivo_Exception	*Error*

public void agregaPedidos(int nroMesa, int cant, int idProd) throws StockInsuficiente_Exception, NoExisteID_Exception

Valores de entrada	Salida esperada	Salida obtenida
{nroMesa de una Mesa, 1, idProd de un Producto}	producto.getStcokInicial() - 1	producto.getStcokInicial()
{nroMesa de una Mesa que no se encuentra en el sistema, 1, idProd de un Producto}	producto.getStcokInicial() - 1	*Error*
{nroMesa de una Mesa, -1, idProd de un Producto}	producto.getStcokInicial() + 1	producto.getStcokInicial()

El contrato no menciona nada sobre nroMesas de Mesas inexistentes. Tampoco menciona nada sobre pasar una cantidad negativa.

boolean verificaPassword (String password)

Valores de entrada	Salida esperada	Salida obtenida
{null}	false	*Error*

El contrato no menciona nada sobre pasarle como parámetro un string null.

- Clase Sistema:

void login(String userName, String password) hrows UserNameIncorrecto_Exception, ContrasenaIncorrecta_Exception, OperarioInactivo_Exception

El contrato no especifica cuándo se lanza OperarioInactivo_Exception. No se puede testear.

- GestionComandas, GestionMesas , GestionMozos, GestionOperario, GestionProdPromo, GestionProdTemp, GestionProductos

  Ningún método tiene contrato, no podemos testearlos.

Caja blanca (cobertura)

En esta parte del trabajo práctico se llevaron a cabo los tests de cobertura de aquellos métodos de las clases FuncionalidadAdmin, FuncionalidadOperario y Sistema en donde los tests de caja negra no llegaron a cubrir todos los caminos posibles.

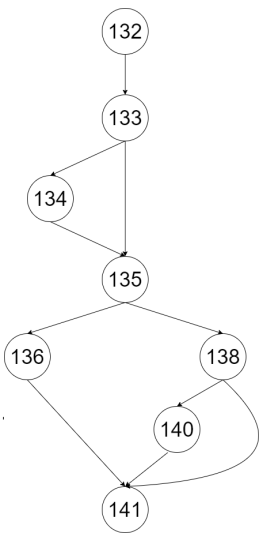
- FuncionalidadAdmin:

void registraOperario

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
129    public void registraOperario(String NyA, String userName, String password, Enumerados.estadoOperario estado)
130            throws UserNameRepetido_Exception, ContrasenaIncorrecta_Exception
131    {
132        boolean activo = true;
133        if (estado == Enumerados.estadoOperario.INACTIVO)
134            activo = false;
135        if (this.verificaPassword(password) == false)
136            throw new ContrasenaIncorrecta_Exception(
137                    "El campo contrase@a debe contener entre 6 y 12 caracteres. Con al menos 1 d@gito y 1 may@scula");
138        else if (Sistema.getInstance().getOperariosRegistrados().putIfAbsent(userName,
139                new Operario(NyA, userName, password, activo)) != null)
140            throw new UserNameRepetido_Exception("El userName '" + userName + "' ya esta asociado a un operario.");
141    }
```

Grafo de control



Complejidad ciclomática = 4.

Caminos

1) 132 - 133 - 135 - 136 - 141
2) 132 - 133 - 135 - 138 - 141
3) 132 - 133 - 135 - 138 - 140 - 141
4) 132 - 133 - 134 - 135 - 136 - 141

Con caja negra ya se cubrieron los caminos 1, 2 y 4

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 3.	Valor de las variables NyA, username, password y estado	- Modificar el valor de username por el username de un Operario existente en el sistema - Modificar el valor de password por una que contenga entre 6 y 12 caracteres con al menos un dígito y una mayúscula	UserNameRepetido_Exception

Código luego de T1:

```
129    public void registraOperario(String NyA, String userName, String password, Enumerados.estadoOperario estado)
130            throws UserNameRepetido_Exception, ContrasenaIncorrecta_Exception
131    {
132        boolean activo = true;
133        if (estado == Enumerados.estadoOperario.INACTIVO)
134            activo = false;
135        if (this.verificaPassword(password) == false)
136            throw new ContrasenaIncorrecta_Exception(
137                    "El campo contrase@a debe contener entre 6 y 12 caracteres. Con al menos 1 d◆gito y 1 may@scula");
138        else if (Sistema.getInstance().getOperariosRegistrados().putIfAbsent(userName,
139                new Operario(NyA, userName, password, activo)) != null)
140            throw new UserNameRepetido_Exception("El userName '" + userName + "' ya esta asociado a un operario.");
141    }
```

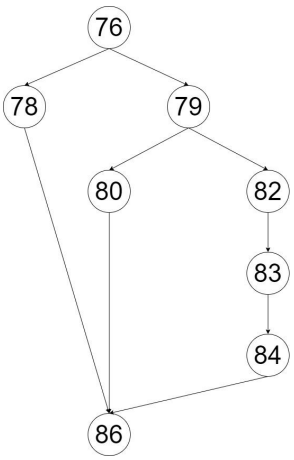
- FuncionalidadOperario:

void modificaOperario

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
74    public void modificaOperario(String NyA, String userName, String password)
75            throws UserNameRepetido_Exception, ContrasenaIncorrecta_Exception {
76        if (userName != this.operario.getUserName()
77                && Sistema.getInstance().getOperariosRegistrados().containsKey(userName))
78            throw new UserNameRepetido_Exception("El nombre de usuario ya se encuentra registrado en el sistema.");
79        else if (this.verificaPassword(password)== false)
80            throw new ContrasenaIncorrecta_Exception("El campo contrase@a debe contener entre 6 y 12 caracteres. Con al menos 1 d◆gito y 1 may@scula");
81        else {
82            this.operario.setNyA(NyA);
83            this.operario.setPassword(password);
84            this.operario.setUserName(userName);
85        }
86    }
```

Grafo de control



Complejidad ciclomática = 3.

Caminos

1) 76 - 78 - 86
2) 76 - 79 - 80 - 86
3) 76 - 79 - 82 - 83 - 84 - 86

Con caja negra ya se cubrieron los caminos 1 y 2

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 3.	Valor de las variables NyA, username y password.	- Modificar el valor de username por un username nuevo en el sistema - Modificar el valor de password por una que contenga entre 6 y 12 caracteres con al menos un dígito y una mayúscula	Se modifican los valores del Operario.

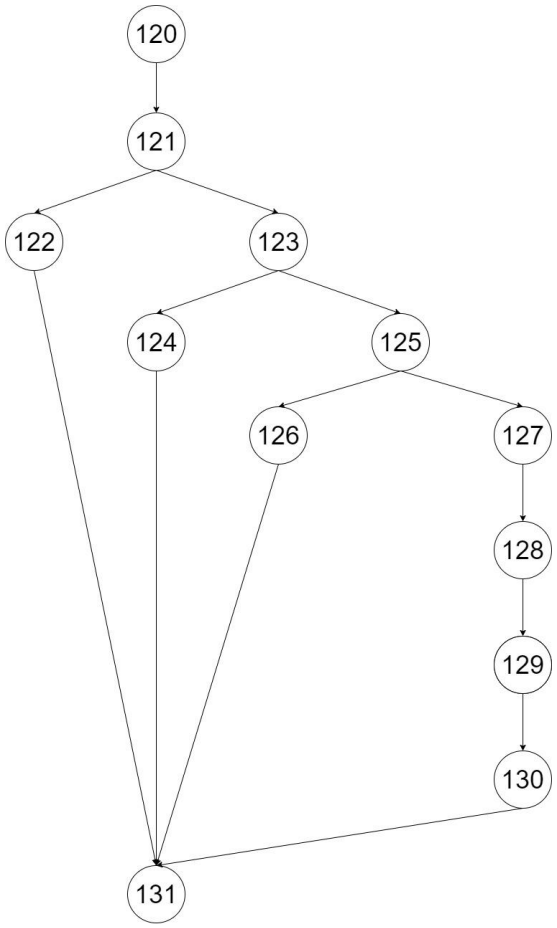
Código luego de T1:

```
74    public void modificaOperario(String NyA, String userName, String password)
75            throws UserNameRepetido_Exception, ContrasenaIncorrecta_Exception {
76        if (userName != this.operario.getUserName()
77                && Sistema.getInstance().getOperariosRegistrados().containsKey(userName))
78            throw new UserNameRepetido_Exception("El nombre de usuario ya se encuentra registrado en el sistema.");
79        else if (this.verificaPassword(password)== false)
80            throw new ContrasenaIncorrecta_Exception("El campo contrase□a debe contener entre 6 y 12 caracteres. Con al menos 1 d◆gito y 1 may□scula");
81        else {
82            this.operario.setNyA(NyA);
83            this.operario.setPassword(password);
84            this.operario.setUserName(userName);
85        }
86    }
```

void modificaProducto

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```java
118    public void modificaProducto(int idProd, String nombre, double precioCosto, double precioVenta, int stockInicial)
119            throws NoExisteID_Exception, precioInvalido_Exception, prodEnUso_Exception{
120        Producto prodActual = Sistema.getInstance().getProductos().get(idProd);
121        if (prodActual == null)
122            throw new NoExisteID_Exception("No existe el producto que desea modificar. Ingrese un ID valido.");
123        if (precioCosto < 0 || precioVenta < 0)
124            throw new precioInvalido_Exception("Ninguno de los precios puede ser negativo.");
125        if (GestionComandas.contieneProd(idProd) == true)
126            throw new prodEnUso_Exception("El producto esta en una comanda activa, no puede ser modificado");
127        prodActual.setNombre(nombre);
128        prodActual.setPrecioCosto(precioCosto);
129        prodActual.setPrecioVenta(precioVenta);
130        prodActual.setStockInicial(stockInicial);
131    }
```



Grafo de control

Complejidad ciclomática = 4.

Caminos

1) 120 - 121 - 122 - 131
2) 120 - 121 - 123 - 124 - 131
3) 120 - 121 - 123 - 125 - 126 - 131
4) 120 - 121 - 123 - 125 - 127 - 128 - 129 - 130 - 131

Con caja negra ya se cubrieron los caminos 1, 2 y 4.

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 3.	Valor de las variables idProd, nombre, precioCosto, precioVenta y stockInicial	- Modificar el valor de idProd por el idProd de un Producto que se encuentre en una comanda activa.	prodEnUso _Exception

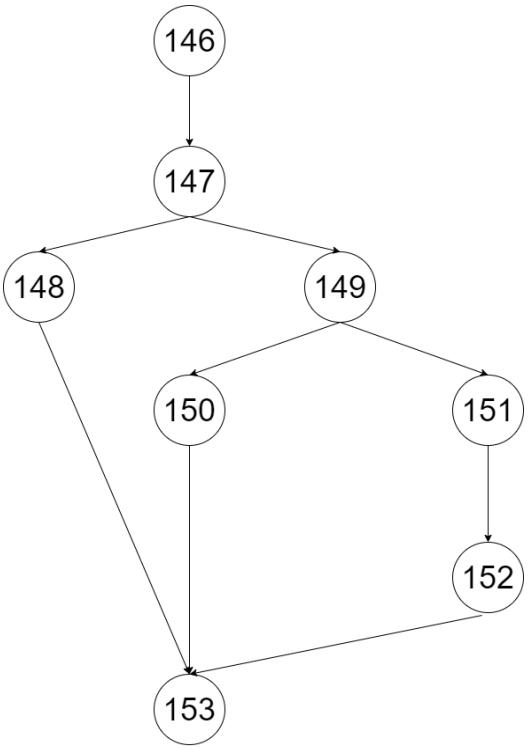
```
118    public void modificaProducto(int idProd, String nombre, double precioCosto, double precioVenta, int stockInicial)
119            throws NoExisteID_Exception, precioInvalido_Exception, prodEnUso_Exception{
120        Producto prodActual = Sistema.getInstance().getProductos().get(idProd);
121        if (prodActual == null)
122            throw new NoExisteID_Exception("No existe el producto que desea modificar. Ingrese un ID valido.");
123        if (precioCosto < 0 || precioVenta < 0)
124            throw new precioInvalido_Exception("Ninguno de los precios puede ser negativo.");
125        if (GestionComandas.contieneProd(idProd) == true)
126            throw new prodEnUso_Exception("El producto esta en una comanda activa, no puede ser modificado");
127        prodActual.setNombre(nombre);
128        prodActual.setPrecioCosto(precioCosto);
129        prodActual.setPrecioVenta(precioVenta);
130        prodActual.setStockInicial(stockInicial);
131    }
```

void modificaMesa

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
144    public void modificaMesa(int nroMesa, int cantPax, Enumerados.estadoMesa estado)
145            throws NoExisteMesa_Exception, CantComensalesInvalida_Exception {
146        Mesa mesaActual = Sistema.getInstance().getMesas().get(nroMesa);
147        if (mesaActual == null)
148            throw new NoExisteMesa_Exception("No existe la mesa que desea modificar.");
149        if (cantPax < 2 && nroMesa > 0)
150            throw new CantComensalesInvalida_Exception("La cantidad de comensales debe ser mayor a uno.");
151        mesaActual.setCantPax(cantPax);
152        mesaActual.setEstado(estado);
153    }
```

Grafo de control



Complejidad ciclomática = 3.

Caminos

1) 146 - 147 - 148 - 153
2) 146 - 147 - 149 - 150 - 153
3) 146 - 147 - 149 - 151 - 152 - 153

Con caja negra ya se cubrieron los caminos 1 y 3

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 2.	Valor de las variables cantPax, nroMesa y estado	- Modificar el valor de cantPax y de nroMesa a 1.	CantComensalesInvalida_Exception

Luego de T1:

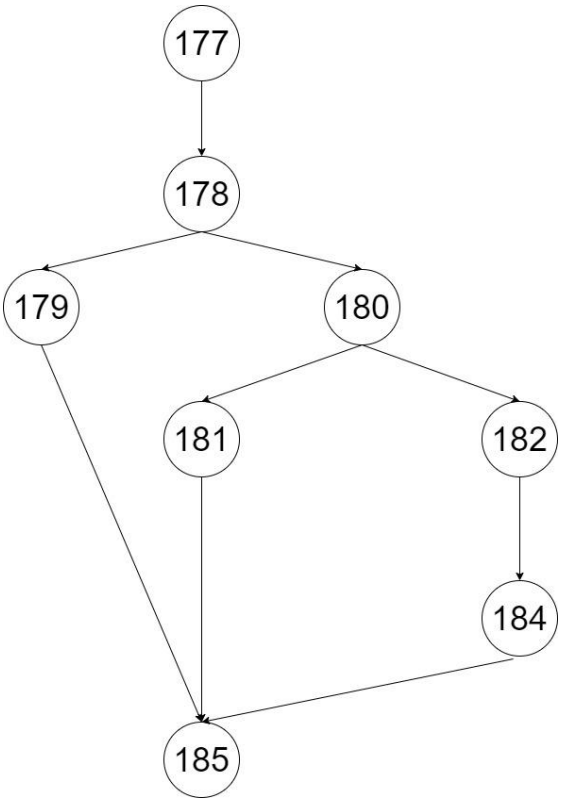
```
144    public void modificaMesa(int nroMesa, int cantPax, Enumerados.estadoMesa estado)
145            throws NoExisteMesa_Exception, CantComensalesInvalida_Exception {
146        Mesa mesaActual = Sistema.getInstance().getMesas().get(nroMesa);
147        if (mesaActual == null)
148            throw new NoExisteMesa_Exception("No existe la mesa que desea modificar.");
149        if (cantPax < 2 && nroMesa > 0)
150            throw new CantComensalesInvalida_Exception("La cantidad de comensales debe ser mayor a uno.");
151        mesaActual.setCantPax(cantPax);
152        mesaActual.setEstado(estado);
153    }
```

void agregaPromocionProd

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
174    public void agregaPromocionProd(int idProd, Enumerados.diasDePromo dia, boolean aplica2x1,
175            boolean aplicaDtoPorCantidad, int dtoPorCantidad_CantMinima, double dtoPorCantidad_PrecioUnitario,
176            boolean activa) throws PromoInvalida_Exception, NoExisteID_Exception {
177        Producto producto = Sistema.getInstance().getProductos().get(idProd);
178        if (producto == null)
179            throw new NoExisteID_Exception("No existe el producto con id " + idProd + ".");
180        if (aplica2x1 == false && aplicaDtoPorCantidad == false)
181            throw new PromoInvalida_Exception("Alguna de las promos 2x1 o dto por cantidad debe ser verdadera.");
182        PromocionProd promoNueva = new PromocionProd(activa, dia, producto, aplica2x1, aplicaDtoPorCantidad,
183                dtoPorCantidad_CantMinima, dtoPorCantidad_PrecioUnitario);
184        Sistema.getInstance().getPromocionProds().put(promoNueva.getIdProm(), promoNueva);
185    }
```

Grafo de control



Complejidad ciclomática = 3.

Caminos

1) 177 - 178 - 179 - 185
2) 177 - 178 - 180 - 181 - 185
3) 177 - 178 - 180 - 182 - 184 - 185

Con caja negra ya se cubrieron los caminos 1 y 3

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 2.	Valor de idProd, dia, aplica2x1, aplicaDtoPorCantidad,	- Modificar el valor de aplica2x1 y de aplicaDtoPorCantidad a false.	PromoInvalida_Exception

		dtoPorCantidad_CantMinima, dtoPorCantidad_PrecioUnitario y activa		

Luego de T1:

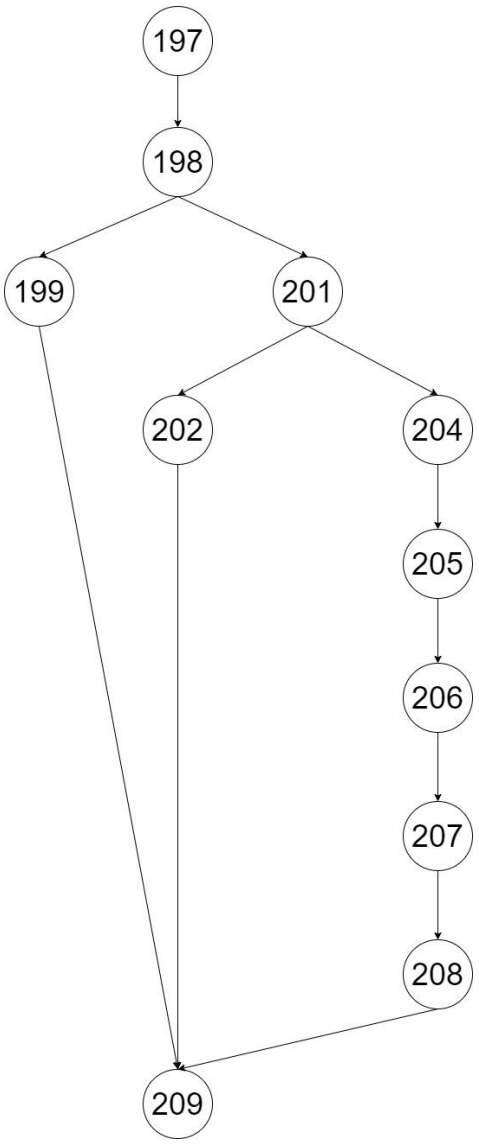
```
173    public void agregaPromocionProd(int idProd, Enumerados.diasDePromo dia, boolean aplica2x1,
174            boolean aplicaDtoPorCantidad, int dtoPorCantidad_CantMinima, double dtoPorCantidad_PrecioUnitario,
175            boolean activa) throws PromoInvalida_Exception, NoExisteID_Exception {
176        Producto producto = Sistema.getInstance().getProductos().get(idProd);
177        if (producto == null)
178            throw new NoExisteID_Exception("No existe el producto con id " + idProd + ".");
179        if (aplica2x1 == false && aplicaDtoPorCantidad == false)
180            throw new PromoInvalida_Exception("Alguna de las promos 2x1 o dto por cantidad debe ser verdadera.");
181        PromocionProd promoNueva = new PromocionProd(activa, dia, producto, aplica2x1, aplicaDtoPorCantidad,
182                dtoPorCantidad_CantMinima, dtoPorCantidad_PrecioUnitario);
183        Sistema.getInstance().getPromocionProds().put(promoNueva.getIdProm(), promoNueva);
184    }
```

void moidificaPromocionProd

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
194    public void modificaPromocionProd(int idProm, boolean activa, Enumerados.diasDePromo dia, boolean aplica2x1,
195            boolean aplicaDtoPorCantidad, int dtoPorCantidad_CantMinima, double dtoPorCantidad_PrecioUnitario)
196            throws PromoInvalida_Exception, NoExisteID_Exception {
197        PromocionProd promoActual = Sistema.getInstance().getPromocionProds().get(idProm);
198        if (promoActual == null)
199            throw new NoExisteID_Exception("No existe la promo " + idProm + ".");
200
201        if (aplica2x1 == false && aplicaDtoPorCantidad == false)
202            throw new PromoInvalida_Exception("Alguna de las promos 2x1 o dto por cantidad debe ser verdadera.");
203
204        promoActual.setAplica2x1(aplica2x1);
205        promoActual.setAplicaDtoPorCant(aplicaDtoPorCantidad);
206        promoActual.setDtoPorCant_CantMinima(dtoPorCantidad_CantMinima);
207        promoActual.setDtoPorCant_PrecioUnitario(dtoPorCantidad_PrecioUnitario);
208        promoActual.setActiva(activa);
209    }
```

Grafo de control



Complejidad ciclomática = 3.

Caminos

1) 197 - 198 - 199 - 209
2) 197 - 198 - 201 - 202 - 209
3) 197 - 198 - 201 - 204 - 205 - 206 -207 - 208 - 209

Con caja negra ya se cubrieron los caminos 1 y 3

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 2.	Valor de idProm, dia, aplica2x1, aplicaDtoPorCantidad, dtoPorCantidad_CantMinima, dtoPorCantidad_PrecioUnitario y activa	- Modificar el valor de aplica2x1 y de aplicaDtoPorCantidad a false.	PromoInvalida_Exception

Luego de T1:

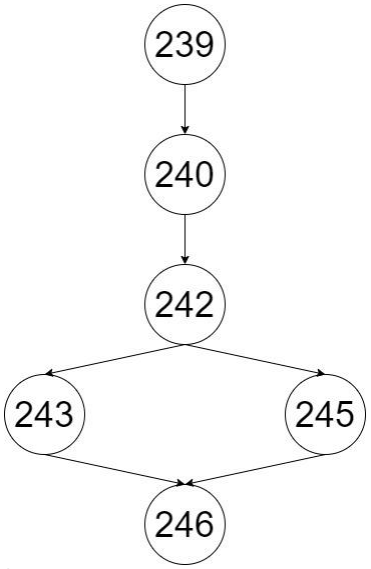
```
193    public void modificaPromocionProd(int idProm, boolean activa, Enumerados.diasDePromo dia, boolean aplica2x1,
194            boolean aplicaDtoPorCantidad, int dtoPorCantidad_CantMinima, double dtoPorCantidad_PrecioUnitario)
195            throws PromoInvalida_Exception, NoExisteID_Exception {
196        PromocionProd promoActual = Sistema.getInstance().getPromocionProds().get(idProm);
197        if (promoActual == null)
198            throw new NoExisteID_Exception("No existe la promo " + idProm + ".");
199
200        if (aplica2x1 == false && aplicaDtoPorCantidad == false)
201            throw new PromoInvalida_Exception("Alguna de las promos 2x1 o dto por cantidad debe ser verdadera.");
202
203        promoActual.setAplica2x1(aplica2x1);
204        promoActual.setAplicaDtoPorCant(aplicaDtoPorCantidad);
205        promoActual.setDtoPorCant_CantMinima(dtoPorCantidad_CantMinima);
206        promoActual.setDtoPorCant_PrecioUnitario(dtoPorCantidad_PrecioUnitario);
207        promoActual.setActiva(activa);
208    }
```

void agregaPromocionTemporal

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
236    public void agregaPromocionTemporal(boolean activa, modelo.Enumerados.diasDePromo diasDePromo, String nombre,
237            modelo.Enumerados.formaDePago formaDePago, int porcentajeDesc, boolean esAcumulable, int horaInicio,
238            int horaFinal) throws PromoRepetida_Exception {
239        ArrayList<PromocionTemporal> promosTemp = Sistema.getInstance().getPromocionesTemp();
240        PromocionTemporal promoActual = new PromocionTemporal(activa, diasDePromo, nombre, formaDePago, porcentajeDesc,
241                esAcumulable, horaInicio, horaFinal);
242        if (promosTemp.contains(promoActual))
243            throw new PromoRepetida_Exception("Ya existe la promo '" + nombre + "'.");
244        else
245            promosTemp.add(promoActual);
246    }
```

Grafo de control



Complejidad ciclomática = 2.

Caminos

1) 239 - 240 - 242 - 243 - 246
2) 239 - 240 - 242 - 245 - 246

Con caja negra se cubrió el camino 2.

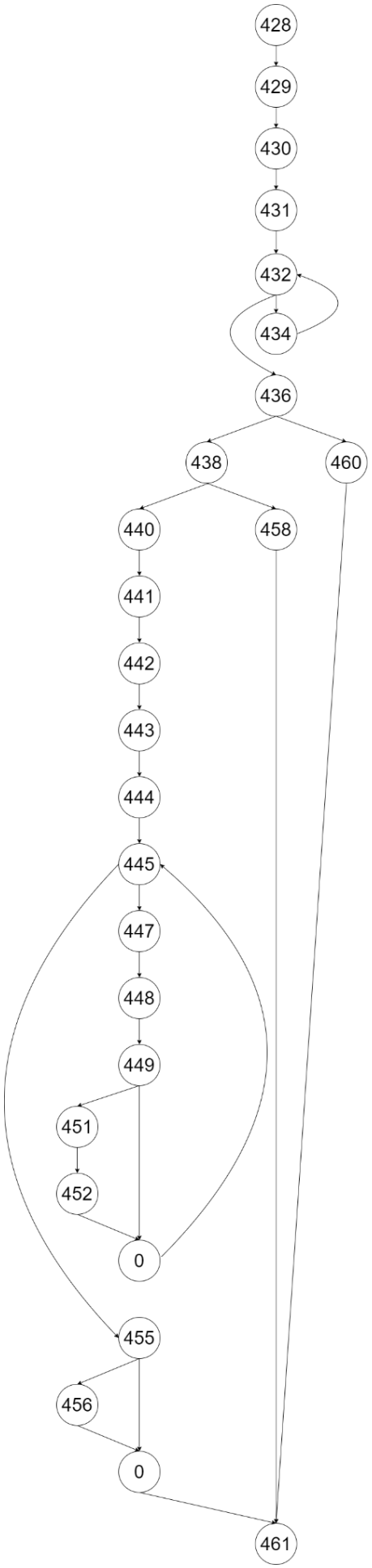
Es imposible realizar el camino 1 debido a que la condición de la línea 242 siempre es falsa. La variable promoActual no puede existir en el Sistema porque es creada en ese momento. No importa que tenga los mismos atributos que otra PromocionProd, siguen siendo dos objetos distintos.

void abreComanda

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
426     public void abreComanda(Mesa mesa) throws NoExisteMesa_Exception, MesaOcupada_Exception, MozoInactivo_Exception
427     {
428         int i = 0, bandera = 0;
429         String nombreMozo = "";
430         HashMap<String, Mozo> mozos = Sistema.getInstance().getMozos();
431         ArrayList<String> arrayMozo = new ArrayList<String>();
432         for (HashMap.Entry<String, Mozo> entry : mozos.entrySet())
433         {
434             arrayMozo.add(entry.getKey());
435         }
436         if (mesa != null)
437         {
438             if (Sistema.getInstance().getMesas().get(mesa.getNroMesa()).getEstado() == Enumerados.estadoMesa.LIBRE)
439             {
440                 Sistema.getInstance().getMesas().get(mesa.getNroMesa()).setEstado(Enumerados.estadoMesa.OCUPADA);
441                 Sistema sistema = Sistema.getInstance();
442                 Comanda comanda = new Comanda(mesa, Enumerados.estadoComanda.ABIERTO);
443                 mesa.setComanda(comanda);
444                 sistema.getComandas().add(comanda);
445                 while (i < 30 && bandera == 0)
446                 {
447                     i = (int) (Math.random() * (arrayMozo.size() + 1));
448                     nombreMozo = arrayMozo.get(i);
449                     if (Sistema.getInstance().getMozos().get(nombreMozo).getEstado() == Enumerados.estadoMozo.ACTIVO)
450                     {
451                         bandera = 1;
452                         mesa.setMozo(Sistema.getInstance().getMozos().get(nombreMozo));
453                     }
454                 }
455                 if (bandera == 0)
456                     throw new MozoInactivo_Exception("No hay mozos activos en este momento");
457             } else
458                 throw new MesaOcupada_Exception("La mesa se encuentra ocupada.");
459         } else
460             throw new NoExisteMesa_Exception("No existe la mesa en el local");
461     }
```

Grafo de control



Complejidad ciclomática = 7.

Caminos

1) 428 - 429 - 430 - 431 - 432 - 436 - 460 - 461
2) 428 - 429 - 430 - 431 - 432 - 434 - 432 - 436 - 460 - 461
3) 428 - 429 - 430 - 431 - 432 - 436 - 438 - 458 - 461
4) 428 - 429 - 430 - 431 - 432 - 436 - 438 - 440 - 441 - 442 - 443 - 444 - 445 - 455 - 0 - 461
5) 428 - 429 - 430 - 431 - 432 - 436 - 438 - 440 - 441 - 442 - 443 - 444 - 445 - 455 - 456 - 0 - 461
6) 428 - 429 - 430 - 431 - 432 - 436 - 438 - 440 - 441 - 442 - 443 - 444 - 445 - 447 - 448 - 449 - 0 - 445 - 455 - 0 - 461
7) 428 - 429 - 430 - 431 - 432 - 436 - 438 - 440 - 441 - 442 - 443 - 444 - 445 - 447 - 448 - 449 - 451 - 452 - 0 - 445 - 455 - 0 - 461

Con caja negra se cubrieron los caminos 1, 2, 3, 4, 6 y 7

Es imposible realizar el camino 5 porque la condición de la línea 456 siempre es falsa. Si ningún mozo está activo, en la línea 448 se salta de los límites del ArrayList. El sistema tiene seis mozos.

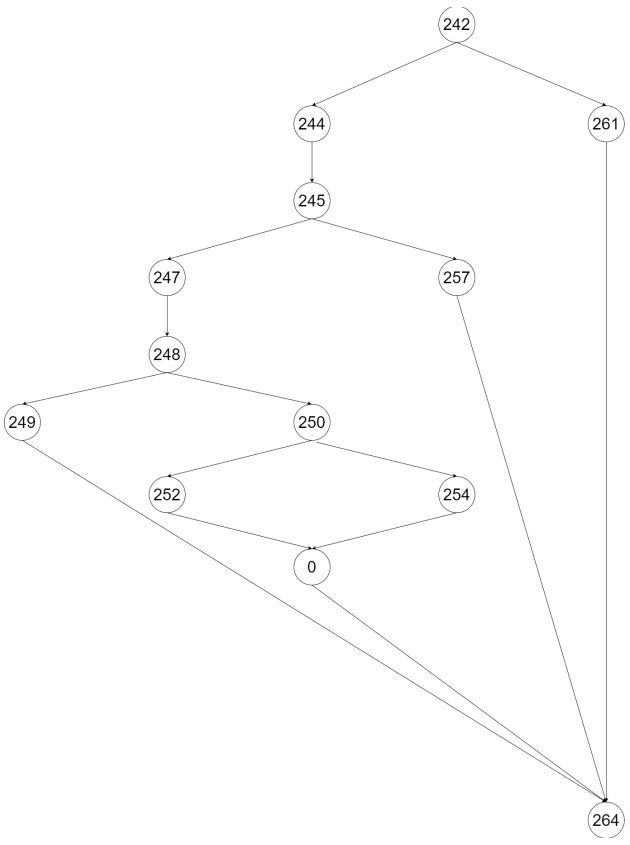
- Sistema:

void login

Luego de aplicar caja negra, las líneas cubiertas son las siguientes:

```
238    public void login(String userName, String password)
239            throws UserNameIncorrecto_Exception, ContrasenaIncorrecta_Exception, OperarioInactivo_Exception
240    {
241        // FuncionalidadOperario funcionalidad = null;
242        if (this.operariosRegistrados.containsKey(userName))
243        {
244            Operario operario = this.operariosRegistrados.get(userName);
245            if (operario.verificaPassword(password) == true)
246            {
247                this.operarioActual = operario;
248                if (operario.isActivo() == false)
249                    throw new OperarioInactivo_Exception("El usuario '" + userName + "' se encuentra inactivo.");
250                if (userName.equals(this.usuarioAdministrador)) // si la contrasena sigue siendo Admin1234 hay que
251                                                                // obligarlo a cambiarla
252                    this.funcionalidadAdmin = new FuncionalidadAdmin(operario);
253                else
254                    this.funcionalidadOperario = new FuncionalidadOperario(operario);
255            } else
256            {
257                throw new ContrasenaIncorrecta_Exception("Contrasena incorrecta.");
258            }
259        } else
260        {
261            throw new UserNameIncorrecto_Exception(
262                    "El usuario '" + userName + "' no se encuentra registrado en el sistema.");
263        }
264    }
```

Grafo de control



Complejidad ciclomática = 5.

Caminos

1) 242 - 261 - 264
2) 242 - 244 - 245 - 257 - 264
3) 242 - 244 - 245 - 247 - 248 - 249 - 264
4) 242 - 244 - 245 - 247 - 248 - 250 - 252 - 0 - 264
5) 242 - 244 - 245 - 247 - 248 - 250 - 254 - 0 - 264

Con caja negra se cubrieron los caminos 1, 2 y 5.

Casos de Prueba

id	Objetivo de la prueba	Datos de entrada	Procedimiento	Salida esperada
T1	Verificar camino 3.	Valor de userName y password	- Modificar el valor de userName al userName de un Operario inactivo.	OperarioInactivo_Exception
T2	Verificar camino 4.	Valor de userName y password	- Modificar el valor de userName al del Administrador.	new FuncionalidadAdmin()

Luego de T1:

```
238    public void login(String userName, String password)
239            throws UserNameIncorrecto_Exception, ContrasenaIncorrecta_Exception, OperarioInactivo_Exception
240    {
241        // FuncionalidadOperario funcionalidad = null;
242        if (this.operariosRegistrados.containsKey(userName))
243        {
244            Operario operario = this.operariosRegistrados.get(userName);
245            if (operario.verificaPassword(password) == true)
246            {
247                this.operarioActual = operario;
248                if (operario.isActivo() == false)
249                    throw new OperarioInactivo_Exception("El usuario '" + userName + "' se encuentra inactivo.");
250                if (userName.equals(this.usuarioAdministrador)) // si la contrasena sigue siendo Admin1234 hay que
251                                                                // obligarlo a cambiarla
252                    this.funcionalidadAdmin = new FuncionalidadAdmin(operario);
253                else
254                    this.funcionalidadOperario = new FuncionalidadOperario(operario);
255            } else
256            {
257                throw new ContrasenaIncorrecta_Exception("Contrasena incorrecta.");
258            }
259        } else
260        {
261            throw new UserNameIncorrecto_Exception(
262                    "El usuario '" + userName + "' no se encuentra registrado en el sistema.");
263        }
264    }
```

Luego de T2:

```
238     public void login(String userName, String password)
239             throws UserNameIncorrecto_Exception, ContrasenaIncorrecta_Exception, OperarioInactivo_Exception
240     {
241         // FuncionalidadOperario funcionalidad = null;
242         if (this.operariosRegistrados.containsKey(userName))
243         {
244             Operario operario = this.operariosRegistrados.get(userName);
245             if (operario.verificaPassword(password) == true)
246             {
247                 this.operarioActual = operario;
248                 if (operario.isActivo() == false)
249                     throw new OperarioInactivo_Exception("El usuario '" + userName + "' se encuentra inactivo.");
250                 if (userName.equals(this.usuarioAdministrador)) // si la contrasena sigue siendo Admin1234 hay que
251                                                                 // obligarlo a cambiarla
252                     this.funcionalidadAdmin = new FuncionalidadAdmin(operario);
253                 else
254                     this.funcionalidadOperario = new FuncionalidadOperario(operario);
255             } else
256             {
257                 throw new ContrasenaIncorrecta_Exception("Contrasena incorrecta.");
258             }
259         } else
260         {
261             throw new UserNameIncorrecto_Exception(
262                     "El usuario '" + userName + "' no se encuentra registrado en el sistema.");
263         }
264     }
```

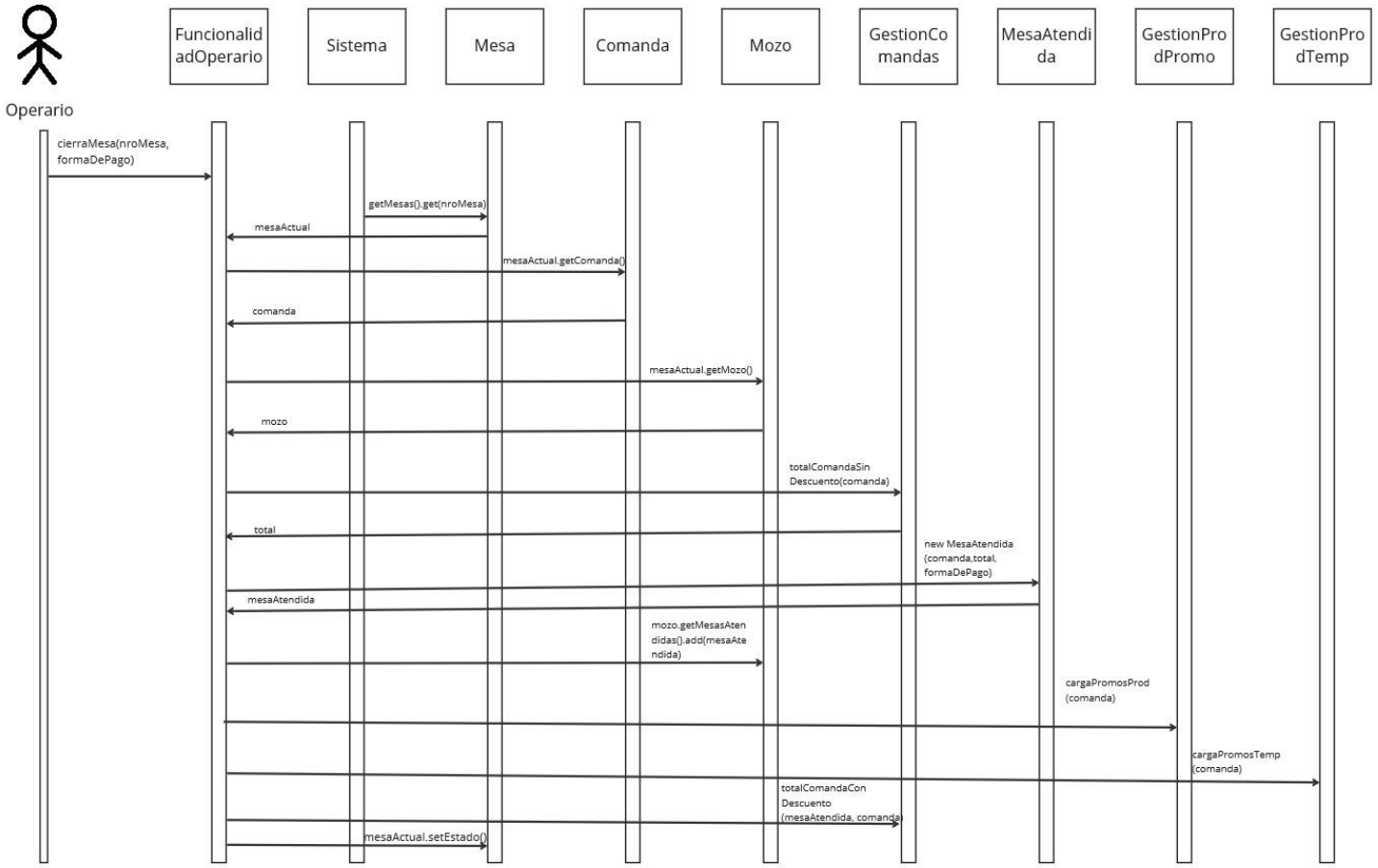
Test de Integración

Las pruebas de integración están diseñadas para probar la interacción entre los distintos componentes de un sistema. En este caso se mostrará el procedimiento seguido para testear el método cierraMesa de la clase FuncionalidadOperario ya que interactúan muchas clases del programa y es una buen ejemplo de integración. De todas formas, todos los demás métodos también fueron testeados y los errores encontrados están en la sección de caja negra.

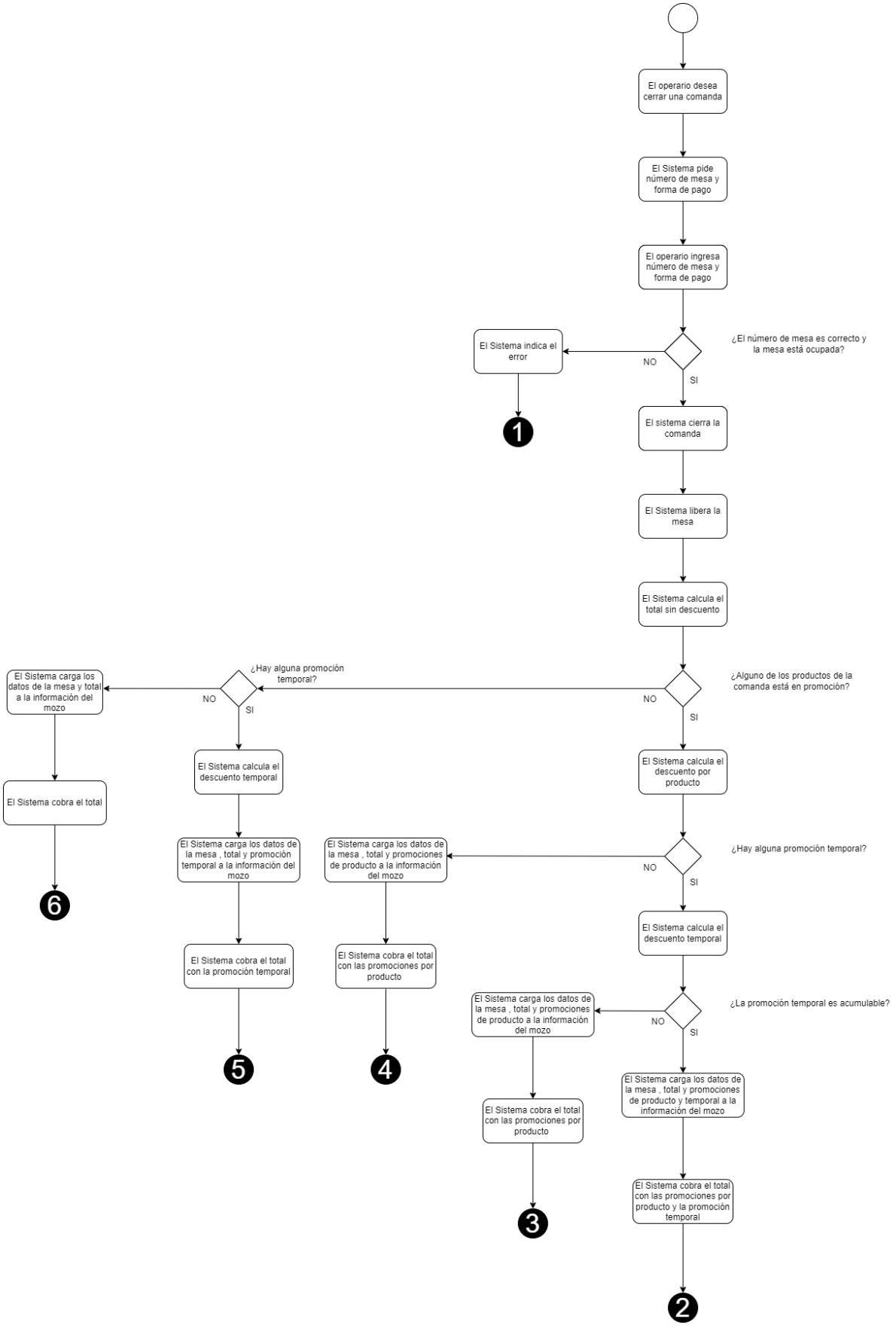
En este caso se aplicó una integración ascendente, lo que significa que primero se testearon las clases más atómicas y se fue subiendo en complejidad, por lo que no fue necesario el uso de mocks.

Aclaración: Este método en específico tiene contradicciones entre el contrato y el código en sí. En el contrato especifica que la excepción MesaNoOcupadaException se lanza cuando la mesa correspondiente al número de mesa pasado como parámetro está ocupada. Sin embargo, en el código hace lo contrario; se lanza cuando la mesa está libre. Es por eso que en el test de caja negra se tomó como verdad únicamente lo estipulado en el contrato (ya que en los tests de caja negra no se debe leer el código). Pero en esta parte del testing se tomó directamente lo que el método hace y no lo que el contrato dice que hace, para poder hacer un testeo integral más adecuado. Así que en este caso se testeó a sabiendas que MesaNoOcupadaException se lanza cuando la mesa está libre.

- Diagrama secuencia: para una mejor visualización hacer click diagrama-secuencia.pdf



- Diagrama de actividad: para una mejor visualización hacer click Diagrama de Actividad.png



En este caso, al ser seis flujos, se identificaron seis casos de uso. Los números debajo de cada flujo en el diagrama indican el caso. A partir de los casos de uso, se llevaron a cabo los siguientes casos de prueba:

Escenarios

Número de escenario	Descripción
1	Sistema con promoción temporal acumulable
2	Sistema con promoción temporal no acumulable
3	Sistema sin promociones temporales

Tabla de particiones

Condiciones de entrada	Clases válidas	Clases inválidas
nroMesa	referente a una mesa en el Sistema que tenga asociada una comanda con un producto en promoción (1)	referente a una mesa fuera del Sistema (2)
	referente a una mesa en el Sistema que tenga asociada una comanda sin productos en promoción (3)	referente a una mesa libre (4)
formaDePago	Enumerados.formaDePago.CTADNI (5)	null (9)
	Enumerados.formaDePago.EFECTIVO (6)	
	Enumerados.formaDePago.MERCPAGO (7)	
	Enumerados.formaDePago.TARJETA (8)	

Batería de pruebas

Escenario 1:

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Flujo recorrido	Salida esperada	Salida obtenida
correcta	{nroMesaConPromProd, Enumerados.formaDePago.CTADNI}	1 y 5	2	Se cierra la mesa aplicando descuento por promoción de producto y por promoción temporal	Se cierra la mesa aplicando descuento por promoción de producto y por promoción temporal
correcta	{nroMesaSinPromProd, Enumerados.formaDePago.	3 y 6	5	Se cierra la mesa aplicando descuento por	Se cierra la mesa aplicando descuento por

	EFECTIVO}			promoción temporal	promoción temporal
incorrecta	{nroMesaFueraDelSistema, Enumerados.formaDePago.MERCPAGO}	2 y 7	1	MesaNoOcupadaException	MesaNoOcupadaException
incorrecta	{nroMesaLibre, Enumerados.formaDePago.TARJETA}	4 y 8	1	MesaNoOcupadaException	MesaNoOcupadaException
incorrecta	{MesaConPromProd, null}	9	2	Se cierra la mesa aplicando descuento por promoción de producto y por promoción temporal	Se cierra la mesa aplicando descuento por promoción de producto y por promoción temporal

Escenario 2:

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Flujo recorrido	Salida esperada	Salida obtenida
correcta	{nroMesaConPromProd, Enumerados.formaDePago.CTADNI}	1 y 5	3	Se cierra la mesa aplicando descuento por promoción de producto	Se cierra la mesa aplicando descuento por promoción de producto
correcta	{nroMesaSinPromProd, Enumerados.formaDePago.EFECTIVO}	3 y 6	5	Se cierra la mesa aplicando descuento por promoción temporal	Se cierra la mesa aplicando descuento por promoción temporal

Podemos notar que cuando en la comanda no hay productos en promoción pero hay alguna promoción temporal, se recorre el flujo 5, sea o no acumulable la promoción temporal.

Escenario 3:

Tipo de clase (correcta o incorrecta)	Valores de entrada	Clases de prueba cubiertas	Flujo recorrido	Salida esperada	Salida obtenida
correcta	{nroMesaConPromProd, Enumerados.formaDePago.CTADNI}	1 y 5	4	Se cierra la mesa aplicando descuento por promoción de producto	Se cierra la mesa aplicando descuento por promoción de producto
correcta	{nroMesaSinPromProd, Enumerados.formaDePago.EFECTIVO}	3 y 6	6	Se cierra la mesa	Se cierra la mesa

Test GUI

Las pruebas GUI son el proceso de probar la interfaz gráfica de usuario de un producto para garantizar que cumple con sus especificaciones. Se optó por testear la ventana “Modifica Producto”. Realizando los test se detectó que no se lanza prodEnUso_Exception y tampoco se sabe cuándo tendría que hacerlo ya que no está en la documentación.
Además, el proyecto debería de soportar cuando se modifican los precios y el precio de venta es menor que el precio de costo y tampoco contempla el caso en el cual el stock sea negativo.

- Test hecho con la clase GuiTestEnabledDisanabled:
  Se detectaron los siguientes errores:
  - testSinNombre: "El botón de confirmar debería estar deshabilitado"
  - testSoloPrecioVenta: "El botón de confirmar debería estar deshabilitado"
  - testVacio: "El botón de confirmar debería estar deshabilitado"
  - testSinPrecioVenta: "El botón de confirmar debería estar deshabilitado"
  - testSoloCostoProducto: "El botón de confirmar debería estar deshabilitado"
  - testSInidProd: "El botón de confirmar debería estar deshabilitado"
  - testSoloStock: "El botón de confirmar debería estar deshabilitado"
  - testSoloNombre: "El botón de confirmar debería estar deshabilitado"
  - testSinStock: "El botón de confirmar debería estar deshabilitado"
  - testSoloidProd: "El botón de confirmar debería estar deshabilitado"
  - testSInCostoProducto: "El botón de confirmar debería estar deshabilitado"

  En todos los casos se permite confirmar los cambios con campos vacíos, lo cuál es incorrecto ya que quedarían variables con valor <null>.

- Test hecho con la clase GuiTestConjuntoConDatos:
  Se detectaron los siguientes errores:
  - testPrecioCostoNegativo: Mensaje incorrecto, debería decir 'Ninguno de los precios puede ser negativo.’

- testPrecioVentaNegativo: Mensaje incorrecto, debería decir 'Ninguno de los precios puede ser negativo.'
- testModificaOk: Mensaje incorrecto, debería decir 'Datos actualizados.'

En los tres casos se esperó un mensaje pero en su lugar se obtuvo <null>.

Test Persistencia

En esta parte del trabajo se testeó la escritura y lectura del archivo persistido, cuando existe y cuando no.

En el código que se está testeando, tanto como para persistir y despersisitr, se realizaron tres métodos. Para persistir abrirOutput(), escribir(Object object) y cerrarOutput(), y para despersistir abrirInput(), leer() y cerrarInput(). En primera instancia se testeó que en la persistencia se cree un archivo. A través del testing se determinó que lo realiza correctamente. Luego, se testeó que se lance una excepción al intentar leer un archivo inexistente. Esto también se determinó que lo realice de forma correcta.
La última parte del testing de persistencia consistió en verificar que la lectoescritura de objetos vacíos y no vacíos se realice de forma adecuada. Al ver la cabecera de los métodos se puede observar que este programa persiste objetos de la clase Object y no de una clase en específico. Es por esto que se testeó la persistencia y despersistencia de objetos de las clases Administrador, Comanda, Mesa, MesaAtendida, Mozo, Operario, Producto, PromocionProd y PromocionTemporal.
En el testing, con ninguna de las clases mencionadas anteriormente se determinó una correcta persistencia y despersistencia, ya sea de objetos vacíos o no vacíos. Esto, revisando el javadoc, se puede atribuir a que ninguna de estas clases sobreescribe los métodos equals y hashcode. Por lo tanto, no hay forma de verificar que el programa persista adecuadamente.

Conclusión

El testing es una parte fundamental en el desarrollo del software y, con las herramientas brindadas por la cátedra, nos fue posible detectar diferentes tipos de errores. En algunos casos la dificultad fue mayor por la escasa documentación que tenían algunos métodos y hubo que considerar un número más grande de casos de prueba.
La realización del trabajo fue instructiva ya que nos ayudó a entender en profundidad los temas vistos durante el cuatrimestre porque aplicamos todos los temas a un mismo proyecto.