Taller de Programación I

Trabajo Práctico Final

Grupo 8

- Testers:

  Priano Sobenko, Bautista

  Rojas, Agustin

- Desarrolladores:

  Errobidart, Facundo

  Savarino, Jeremias

Contenidos

Resumen

En el presente trabajo se busca realizar el testing de un sistema informático de administración gastronómica. Para esto, se le entregó al alumnado un documento de requerimientos completo con el fin de construirlo y posteriormente entregar a otro grupo el sistema completo con su respectiva documentación. El objetivo perseguido es aplicar todas las herramientas trabajadas a lo largo del cuatrimestre, para fijar su aprendizaje acercándose lo máximo posible a una situación real.

Introducción

El testing ha abarcado distintas visiones a lo largo de la historia variando de manera conjunta con el desarrollo de la industria del software. La visión más actual dice que el testing es “El proceso de verificar que el software se ajusta a los requerimientos y validar que las funciones se implementan correctamente.” Esta es una de las más completas ya que no implica simplemente que su funcionamiento sea correcto, sino que además lo que se haya desarrollado se ajuste a lo que haya sido planteado por el o los Stakeholders. Además, se percibe a “el hacer pruebas” no como una actividad posterior al desarrollo sino como un proceso paralelo a la construcción del sistema.

Este último aspecto es clave ya que es posible detectar errores e incertezas de manera temprana, para poder realizar las correcciones de manera continua. De esta forma, no se corre el riesgo de generar nuevas fallas que dificultan el desarrollo al intentar solucionar problemas en etapas finales. Esto genera un cambio radical en el tiempo y el esfuerzo invertido en el proyecto, por lo tanto, tendrá un impacto económico muy grande también. Por otra parte, es importante aclarar que ningún método de testing nos proporcionará un 100% de seguridad de que el sistema está libre de errores.

El testing se puede separar en distintos tipos dependiendo de que se prueba, de que conocemos sobre el sistema y el grado de automatización.

Según de qué se prueba:

- Pruebas unitarias. Diseñadas para probar una parte pequeña y específica de funcionalidad. Ej: un método.
- Pruebas de integración. Diseñadas para probar la interacción entre los distintos componentes de un sistema.
- Pruebas de sistema. Diseñadas para probar el sistema en su totalidad como si de una caja negra se tratase.
- Pruebas de aceptación. Diseñadas para verificar que el sistema cumple con los requisitos exigidos por el usuario.

Según de lo que se conoce:

- Caja negra: No se conoce el código, se basa en el contrato.
- Caja blanca: Se conoce el código y el contrato.

Según el grado de automatización:

- Manuales.
- Automáticas.

Se propone realizar las pruebas correspondientes a Test Unitario de Caja Negra, Caja Blanca, Test de Persistencia de datos, Test de Interfaces Gráficas mediante el uso de la clase “Robot”, y, por último, Test de Integración.

Desarrollo

Caja Negra

Los tests de caja negra se destacan por que el tester no cuenta con el código de lo que está probando. Esto quiere decir que la prueba se lleva a cabo con desconocimiento del funcionamiento interno, debido a que se enfoca en la entrada y salida. Toda la información con la que uno cuenta para realizar un testeo proviene de la documentación, a partir de esta podemos construir casos de prueba, para verificar que una cierta entrada produce la salida correcta.

Se utilizan normalmente para testear el elemento mínimo del lenguaje utilizado ya que uno no puede saber la naturaleza del error. De esta forma, se aíslan los errores, de modo que se puedan hallar incongruencias fácilmente.

Para implementar este tipo de pruebas en el proyecto se utilizaron las dependencias de JUnit, ya que nos permite generar de manera relativamente sencilla los casos de prueba y contienen a los métodos de prueba. Además, previamente ejecuta un método llamado “Set up” que setea el escenario requerido y posteriormente ejecuta otro denominado “TearDown” que deja las estructuras de datos tal cual estaban antes de hacer el testeo.

La documentación no fue la óptima para realizar este tipo de testeos ya que limitaba la mayoría de las veces el conocimiento aplicado para llevar a cabo este tipo de práctica. Esto demuestra la importancia de la documentación y que la cantidad de métodos que podamos testear está directamente ligada a la calidad y cantidad de información del código con la que contemos. Una vez elegidos los métodos se eligieron distintos escenarios que modelan lo que ocurre en el sistema mientras se aplican distintos casos de prueba. El resultado dependerá de la combinación de los distintos escenarios y casos de prueba.

Se decidió estudiar profundamente el método:

public PromoFijaDTO agregarPromoFija(PromoFijaRequest request) throws MalaSolicitudException

Escenarios

Escenario = 1	Set<Promociones> vacio
Escenario = 2	Set<Promociones con al menos una promocion del tipo: PromocionFija{“Promo1 ”,dias,Pizza,2x1=true,Desc=false,null,nul}

Para construir los datos de prueba se utilizó el método de particiones de equivalencia. En este se divide el campo de entrada de un programa en clases de equivalencia de donde se pueden derivar casos de prueba. Por contrato no se evalúa la existencia de los productos en el sistema.

Particiones de equivalencia

Condición de Entrada	Clase válida	Clase inválida
request	- nombre sin repetir (1.1) - dias.size() > 0 (2.2) - 2x1 o DescPorCant deben ser opuestos (1.3)	- nombre Repetido (2.1) - dias.size() = 0 (2.2) - botones 2x1 y Desc tienen el mismo valor booleano (2.3)

Batería De Pruebas

Tipo de Clase	Escenario	Entrada	Salida	Salida esperada	Clase Cubierta
Válida	1	PromocionFija { “Test promo”, dias { Domingo, Lunes} , prod:“Test”, 2x1=true, Desc=false, null, nul }	PromofijaDTO	PromoFijaDTO	(1.1) (1.2) (1.3)
Válida	2	PromocionFija { “Promo1”, dias { Domingo, Lunes, Martes} , prod:“Pizza”, 2x1=true, Desc=false, null, nul }	PromofijaDTO	Exception Promocion existente	(2.1) (1.2) (1.3)
Invalida	1	PromocionFija { “Test promo”, dias { } , prod:“Test”, 2x1=true, Desc=false, null, nul }	MalaSolicitud Exception	MalaSolicitud Exception	(1.1) (1.3) (2.2)
Invalida	1	PromocionFija { “Test promo”, dias { Domingo, Lunes} , prod:“Test”, 2x1=false, Desc=false, null, nul }	MalaSolicitud Exception	MalaSolicitudExc eption	(1.1) (1.2) (2.3)
Invalida	1	PromocionFija { “Test promo”, dias { Domingo, Lunes} ,	MalaSolicitud Exception	MalaSolicitudExc eption	(1.1) (1.2) (2.3)

		prod:“Test”, 2x1=true, Desc=true, null, nul }			

Si bien el análisis realizado se vio dificultado por la falta de información se pudo percibir el funcionamiento de la clase. A partir de esto pudimos generar los datos necesarios para completar el método y arrojar que hay un error. El sistema no realiza la validación si la promoción a agregar ya existe en el sistema, demostrado por una falla en el caso de prueba 2 con el escenario 2. Por lo tanto, se concluye que la prueba fue exitosa.

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.

Aunque las pruebas de caja blanca son aplicables a varios niveles (unidad, integración y sistema), habitualmente se aplican a las unidades de software. Su cometido es comprobar los flujos de ejecución dentro de cada unidad (función, clase, módulo, etc.) pero también pueden testear los flujos entre unidades durante la integración, e incluso entre subsistemas, durante las pruebas de sistema.

El objetivo de la técnica es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa, y todas las condiciones tanto en su vertiente verdadera como falsa.

Firma del método:

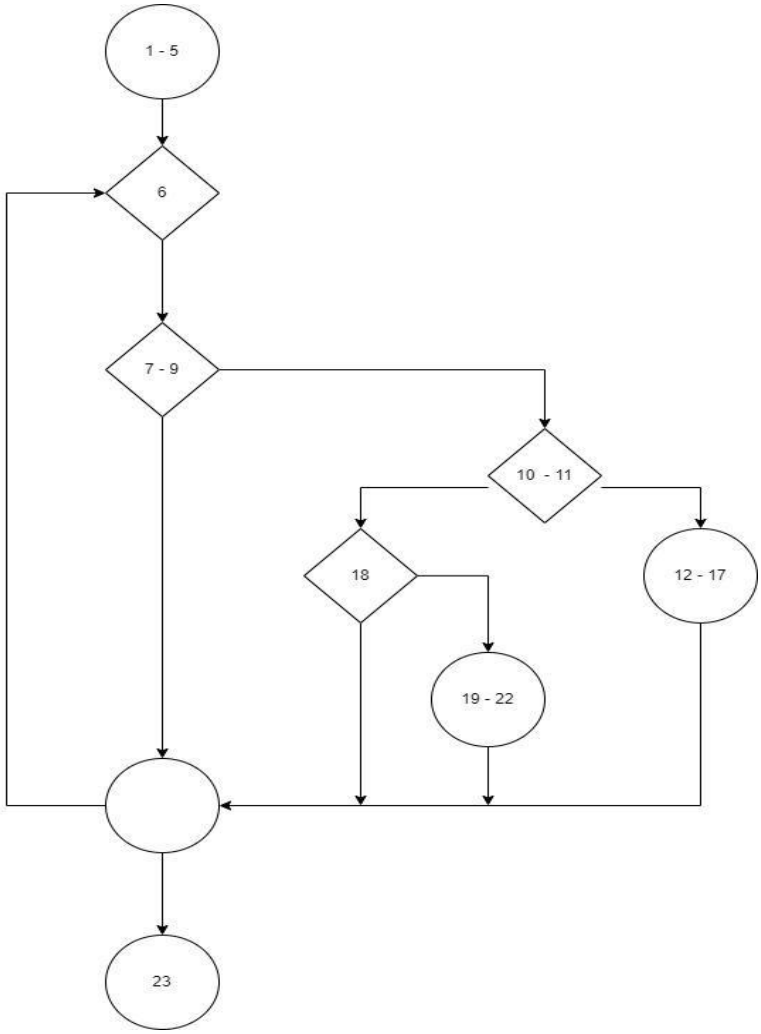
aplicarPromocionesFijas(CierreDeComanda cierreDeComanda);

Ubicación del método:

app-cerveceria/src/main/java/com/grupo8/app/negocio/GestionDeMesas

Este método devuelve un boolean que indica si al cierre de comanda pasado por parámetro, corresponde aplicarle una promoción por producto o no.

Grafo de flujo



Complejidad Ciclomática

= Regiones internas + Región externa
= 5 + 1
= 6

Selección de caminos

Se utilizó el método de construcción general.

- C1: (1-5) , 6 , (7-9) , (10-11) , 18 , (19-22) , 23
- C2: (1-5) , 6 , (7-9) , (10-11) , 18 , 23
- C3: (1-5) , 6 , (7-9) , (10-11) , (12-17) , 23
- C4: (1-5) , 6 , (7-9) , 23
- C5: (1-5) , 6 , 23

Con estos 5 caminos seleccionados se logrará una cobertura total sobre el algoritmo.

Propusimos crear 5 métodos de testeo, cada uno con los valores de entrada necesarios y coherentes para cubrir la totalidad de los caminos.

Escenarios

- *Escenario 1*: Set de promociones fijas vacío.

- *Escenario 2*: Set de promociones fijas con al menos una promoción 2x1.
  Promoción Fija presente en la colección:
  { id: 1 , producto1 , dosPorUno: true, cantidadMinima: null }

- *Escenario 3*: Set de promociones fijas con al menos una promoción aplicable por cantidad mínima de un producto.
  Promoción Fija presente en la colección:
  { id: 1 , producto1 , dosPorUno: false , cantidadMinima: 2 }

Camino	Escenario	Caso	Salida Esperada

C1	3	CierreComanda {
		listaPedidos = [
		{prod: Pizza, cantPedida: 4} ,
		{prod: Fideos, cantPedida: 1}
		]
		nroMesa = 5
		…
		}
			aplicoPromo = true

C2	3	CierreComanda {
		listaPedidos = [
		{prod: Pizza, cantPedida: 2} ,
		{ prod: Fideos, cantPedida: 1}
		]
		nroMesa = 5
		…
		}
			aplicoPromo = false

C3	2	CierreComanda {
		listaPedidos = [
		{prod: Pizza, cantPedida: 3} ,
		{prod: Fideos, cantPedida: 1}
		]
		nroMesa = 5
		…
		}
			aplicoPromo = true

C4	3	CierreComanda{
		listaPedidos = [
		{prod: Sopa, cantPedida: 2} ,
		{prod: Fideos, cantPedida: 1}
		]
		nroMesa = 5
		…
		}
			aplicoPromo = false

C5	1	CierreComanda {
		listaPedidos = [
		{prod: Pizza, cantPedida: 3} ,
		{prod: Fideos, cantPedida: 1}
		]
		nroMesa = 5
		…
		}
			aplicoPromo = false

Se pudo recorrer todos los caminos posibles variando los casos de prueba, esto queda demostrado con la cobertura total del método, indicando que no se hallaron errores de diseño. Por lo tanto, se concluye que la técnica aplicada no halló errores de diseño (Ver siguiente imagen).

```
    /**
     * Aplica las promociones fijas vigentes a una comanda
     * @param cierreComanda la comanda a la cual se le aplicaran las promociones
     * @return booleano que indica si se aplico alguna promocion
     */
    public boolean aplicarPromocionesFijas(CierreComanda cierreComanda) {
        boolean aplicoPromo = false;
        List<Promocion> promos = new ArrayList<>();
        promos.addAll(this.empresa.getPromocionesFijas().getPromocionesFijas());

        List<Promocion> promosFijas =
                promos
                .stream()
                .filter(promo -> promo.getDiasPromo() != null && promo.getDiasPromo().contains(LocalDate.now().getDayOfWeek()) && promo.isActivo()
                .collect(Collectors.toList());


        for (Promocion promo : promosFijas) {
            PromocionFija promoFija = (PromocionFija) promo;
            Optional<Pedido> pedidoDePromo = cierreComanda.getPedidos().stream()
                    .filter(p -> Objects.equals(p.getProducto().getId(), promoFija.getProducto().getId())).findFirst();

            if (pedidoDePromo.isPresent()) {
                Pedido pedido = pedidoDePromo.get();
                if (promoFija.getDosPorUno()) {
                    int pares = Math.floorDiv(pedidoDePromo.get().getCantidad(), 2);
                    int resto = pedidoDePromo.get().getCantidad() % 2;
                    pedido.setSubtotal(pares * pedido.getProducto().getPrecio() + resto * pedido.getProducto().getPrecio());
                    pedido.setEsPromo(true);
                    cierreComanda.getPromociones().add(promoFija);
                    aplicoPromo = true;
                } else { //es una promo por cantidad
                    if (pedido.getCantidad() >= promoFija.getDtoPorCantMin()) {
                        pedido.setSubtotal((float) (pedido.getCantidad() * promoFija.getDtoPorCantPrecioU()));
                        pedido.setEsPromo(true);
                        cierreComanda.getPromociones().add(promoFija);
                        aplicoPromo = true;
                    }
                }
            }
        }

        return aplicoPromo; //Avisa si realmente logro aplicar alguna promo
    }
```

Test de Persistencia de Datos

Los datos en tiempo de ejecución del programa son efímeros, es decir al finalizar la ejecución estos se borran. El objetivo principal de la persistencia es guardar esta información para que él mismo la pueda utilizar en el momento de volverse a ejecutar. Además, la puede utilizar para generar reportes o exportarla para que pueda ser utilizada por otro programa.
Se pueden generar tres tipos de problemas:

- El archivo cuyo nombre suministró el usuario no existe.
- Hay un error en el archivo, lo cual nos impide leerlo (el periférico de almacenamiento está dañado).
- La información del archivo no se encuentra en el formato pedido.

En este trabajo, se desarrollaron dos métodos para testear este proceso.
El primero realizaba la prueba si el método persistir() perteneciente a la clase Gestión De Productos creaba el archivo correspondiente al ejecutase. Simplemente se preguntó si el archivo existe luego de ejecutar el método.

El segundo probaba que el método cargarProductos() perteneciente a la clase “Empresa“ cargue correctamente y despersista los productos serializados del archivo xml. Para esto se generó un set vacío, se lo persistió con el método testeado previamente, luego se despresistió con el método que se está evaluando y finalmente se comparó el set obtenido de la despersistencia con uno vacío (set esperado). No se pudo evaluar la despersistencia sin archivo ya que por contrato la excepción se atrapa antes de llegar al método.

Luego de llevar a cabo dichas implementaciones, los test no lograron encontrar ninguna falla.

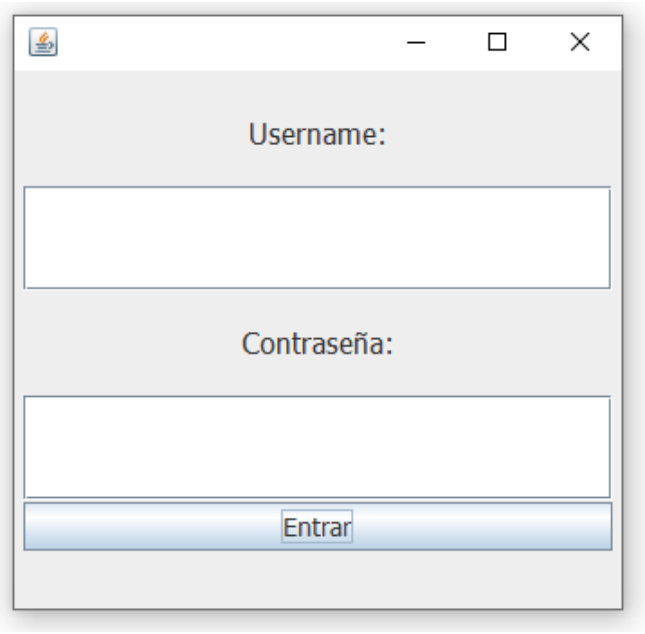
Test de interfaces de usuario

El fin de los testeos de interfaces de usuario es verificar que las funciones de la interfaz gráfica sean correctas y funcionen de la manera esperada. Son unas de las prácticas más complejas del testing, esto se debe a que se combinan las reglas propias del modelo, y el propio comportamiento de la GUI, con sus métodos y atributos.

La dificultad recae en que hay una gran combinación de situaciones posibles, el usuario puede clickear en cualquier pixel de la pantalla y cada componente tiene una cantidad enorme de atributos y métodos.

Para realizar este testeo se eligió una clase que no represente una gran complejidad y se pueda construir un método que tome los distintos caminos y dispare los eventos necesarios.

La interfaz grafica de usuario (GUI) seleccionada para realizar su testeo fue “VistaLogin”



Testeos realizados

Testing del botón "Entrar" para observar su activación o desactivación correspondiente con los datos ingresados

Para ello hemos creado la clase "GuiTestEnabledDisabled" con los métodos necesarios para probar los siguientes casos:

1. Ingresar únicamente el nombre dentro del TextField username.
2. Ingresar únicamente la contraseña dentro del TextField contraseña.
3. TextField username y contraseña vacíos.
4. TextField username y contraseña llenos.

Luego de realizados los testeos correspondientes con la clase Robot sobre estos casos de prueba no hemos detectado fallas. El botón "Entrar" se activa y se desactiva de forma correcta y coherente.

Resultados obtenidos del testing sobre el botón "Entrar"

1. Solo TextField username => botón "Entrar" disabled ✔
2. Solo TextField contrasena => botón "Entrar" disabled ✔
3. TextField username y contrasena vacíos => botón "Entrar" disabled ✔
4. TextField username y contraseña llenos => botón "Entrar" enabled ✔

Testing sobre el ingreso de datos y lanzamiento de OptionPanes

Para ello hemos creado la clase "GuiTestIngresoDatos" con los métodos necesarios para probar los siguientes casos:

1. TextField username y contrasena llenos.
2. TextField username y contrasena vacíos.

Resultados obtenidos del testing sobre el ingreso de datos

1. TextField username y contrasena llenos => login exitoso ✔
   Un assert corrobora que el nombre del usuario logueado sea el correcto.

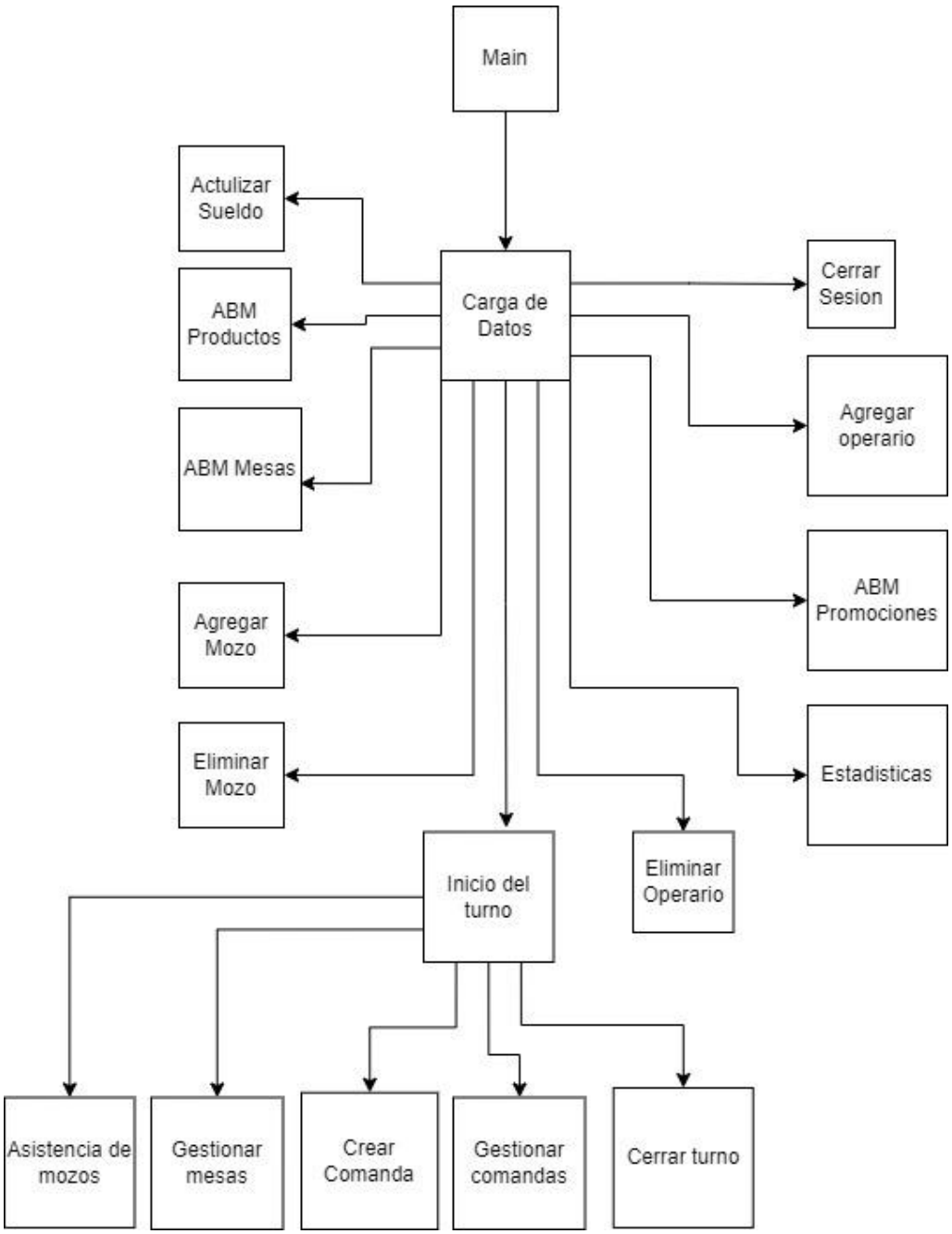
2. TextField username y contrasena vacíos => OptionPane lanzado ✔
   Un assert corrobora que el texto que tiene el panel sea el correcto.

Test de integración

El objetivo de este tipo de práctica es encontrar errores en la interacción entre las distintas unidades o errores de interoperabilidad.

Existen distintos modelos para estas pruebas. Para este caso se decidió utilizar pruebas de tipo incrementales con una estrategia de integración descendente (“Top-Down”), y en profundidad (“Depth-first”).

Utilizamos este modelo debido a la estrategia que optamos para conseguir el caso de uso principal dentro de un flujo típico en el programa. Se construyó un gráfico para poder percibir visualmente este flujo y armar las secuencias de ejecución de forma más simple.



Al mirar rápidamente el gráfico uno puede darse cuenta de la cantidad de distintos caminos que puede tomar el programa, por lo tanto, se diseñaron dos secuencias de ejecución que se entendieron como esenciales para el funcionamiento.

La primera, crea una mesa, crea un mozo y asigna el mozo a la mesa, luego se eliminan ambos. La segunda, abre el local, crea una comanda con la mesa creada, agrega un pedido a la comanda, cierra la comanda y finalmente, cierra el local. En ninguno de los dos casos tuvimos que construir mocks ya que el código recibido está completo.

Al ejecutar el primer caso se realizó de forma exitosa sin arrojar error alguno. En cambio, el segundo método detectó una falla en el paso final, arrojando excepciones constantemente. Al revisar el código para averiguar el origen de la falla se descubrió un error en el constructor de PedidoRequest. Esto generaba que el cierre de la comanda tenga un Id distinto a la comanda que había sido enviada por parámetro, por lo tanto, el cierre y la comanda nunca iban a coincidir. De esta manera, habiendo encontrado una falla en el funcionamiento general del sistema, se concluye que la práctica fue exitosa.

Conclusión

A lo largo del trabajo se aplicaron todas las metodologías de testing dinámicas vistas en la práctica descubriendo errores que sin este tipo de técnicas hubiera sido de gran dificultad descubrirlas. Cada técnica realizó un aporte diferente al conocimiento total del sistema.

Se realizó un *Test de caja negra* de muchos métodos de la capa de negocio del sistema basándose en el contrato recibido. Es un método útil para reconocer grandes fallos, pero se ve afectado cuando la documentación del programa es escasa, ya que dificulta la determinación de entradas, salidas, y posibles errores en la ejecución. Luego, se eligió un método para estudiar en profundidad. Se construyeron escenario, clases validas e invalidas y finalmente juntando esta información, la batería de pruebas. Este halló un error de diseño interesante, el sistema no arroja una excepción si se llenaban todos los campos de una promoción fija exactamente igual a otra que ya estaba en el sistema. De otra forma, hubiera sido muy difícil encontrar el error.

Se llevó a cabo un *Test de caja blanca* en el cual, a partir de un grafo elegido, se construyó un grafo ciclomático de su secuencia de ejecución. Con el grafo, se pudo calcular la máxima cantidad de caminos necesarios para recorrerlo en su totalidad y se construyeron los caminos. Finalmente, se armaron escenarios y casos de prueba para ejecutar cada camino. Este método no arrojó errores, y se logró una cobertura total sobre el algoritmo.

Al realizar el *Test de persistencia* se tuvieron en cuenta las tres principales fallas que puede encontrar esta práctica. Se tuvo en cuenta, como caso de prueba, la creación del archivo y la despersistencia de los datos. Estas pruebas no arrojaron errores.

El *Test de GUI* se realizó sobre la ventana del LogIn debido a su mediana complejidad debido a lo engorrosos que pueden llegar a ser este tipo de tests. Para llevarlo a cabo, con la ayuda y aplicación de la clase “Robot”, se evaluaron dos tipos de casos. El primero, se testea el botón “Entrar” donde se observa la activación o desactivación del botón dependiendo de los datos ingresados. El segundo testea el ingreso de datos y lanzamiento de los distintos OptionPanes dependiendo de la excepción que debería ser lanzada. El método no arrojó errores.

Finalmente, se realizó un *Test de integración* que evalúa la interacción entre varios distintos componentes del sistema. Se evaluaron dos posibles caminos mezclando casos de uso. El test arrojó un error en el constructor de una de las clases.

Para concluir, al tomar como ejemplo este último caso, podemos ver que si evaluamos al terminar el programa sin testear a fondo método por método los errores nos pueden costar mucho tiempo y esfuerzo que tendríamos que invertir en re-trabajo. De esta forma queda evidenciada la importancia del testing en el proceso del desarrollo de software y que esta sea en conjunto con el desarrollo. Se lograron aplicar todas las prácticas estudiadas a lo largo de la cátedra y en varias de ellas se encontraron diversos errores, por lo tanto, se concluye que el trabajo fue exitoso.