Trabajo Práctico Final

Materia: Taller de Programación I

Integrantes:

Suarez Quilis Ignacio

Matarazzo Tomas

Integrantes del otro subgrupo:

Matias Angelico

Lautaro Bruses

Grupo: 1

Fecha de entrega: 16/11/2022

Github: https://github.com/MouGliXx/taller-de-programacion-I/tree/elRealTestingPa

Video:https://www.youtube.com/watch?v=8LILNYZsctI&ab_channel=IgnacioSuarez

Introducción

El testing es una técnica utilizada en el desarrollo de software para evaluar la funcionalidad de una aplicación con la intención de determinar si su desarrollo cumple con los requisitos especificados, permitiendo reconocer y resolver los distintos errores dentro de un sistema informático. El producto de software debidamente probado garantiza la confiabilidad, la seguridad y el alto rendimiento, lo que además se traduce en ahorro de tiempo, rentabilidad y mejoras en la satisfacción del cliente, además de mejorar la experiencia del mismo.

La implementación de las distintas técnicas de testing es valiosa porque si hay errores o bugs en el software, se pueden identificar temprano y se pueden resolver antes de la entrega del producto de software. Es importante porque los errores de software pueden ser costosos o incluso peligrosos, estos pueden potencialmente causar pérdidas monetarias , la historia está llena de ejemplos de este tipo.

El objetivo del informe fue aplicar las distintas técnicas de testing como Caja Negra, Caja Blanca, Test de Integración , Persistencia y Test de GUI a un sistema de cervecería creado por los otros dos integrantes del grupo.

Desarrollo

Test de caja negra

Son aquellos test donde no tenemos información sobre el código, los casos de prueba se basan únicamente en el comportamiento de entrada y salida tomando como base sus especificaciones y requisitos. Las mismas resultan de gran utilidad para la detección de errores de interfaz, de rendimiento, de inicio y terminación.

La clase Cervecería es la que contiene gran parte de los métodos desarrollados en el programa, por lo que decidimos testear métodos que pertenecieran a dicha clase. Se eligieron los siguientes métodos representativos.

Metodo agregarMozo

Si bien se trata de un método que no posee grandes cantidades de código, la elección del mismo estuvo acompañada de la cantidad de estructuras de condición que posee. Además al lanzar 4 excepciones del mismo tipo pero con distinto mensaje, resultó de utilidad para testear el lanzamiento de las mismas. Debido a que por precondición la colección de mozos nunca sería null, planteamos los siguientes escenarios.

Escenario:

1. Escenario en el que la colección mozo se encuentra con mozos.
2. Escenario en el que la colección de mozos se encuentra completa.

Condición de entrada	**Clases correctas**	**Clases incorrectas**
nombre	x = {String valido} 1.	x = {String vacío}
edad	x = { edad >= 18} 3.	x = {edad < 18} 4.
hijos	x = { hijos >= 0} 5.	x = {hijos < 0 } 6.
Estado coleccion mozos	x={vacía o con elementos pero no llena} 7.	x={colección llena} 8.

Tipo de clase	**Valores de entrada**	**Resultado esperado**	**Clases de prueba cubiertas**
Correcta	(“Tomas”, 18 , 0) Escenario 1	Se agrega el nuevo mozo al array de mozos	1. 2. 4. 6.
Incorrecta	(“Tomas” , 18 , 2) Escenario 2	Se recibe excepción, se llegó al máximo de mozos	7.
Incorrecta	(“Tomas”, 15 , 2) Escenario 1	Se recibe excepción, la edad del mozo es menor a 18	3.
Incorrecta	(“Tomas”, 20,-1) Escenario 1	Se recibe excepción, la cant de hijos es menor que cero	5.

Método eliminarComanda

Escenario:

Único escenario en el que el array de comanda es != null

Condición de entrada	Clases correctas	Clases incorrectas
Comanda	x = {Comanda con estado abierta} 1.	x = {Comanda con estado cerrado } 2.

Tipo de clase	Valores de entrada	Resultado esperado	Clases de prueba cubiertas
Correcta	Comanda {“10/11/2022”, Mesa , pedidos , “abierta”}	Se elimina la comanda del array comandas, la Mesa que da en estado libre y se cambia el estado de la comanda a cerrada	1.
Incorrecta	Comanda {“10/11/2022”, Mesa , pedidos , “cerrada”}	Se recibe excepción, no se puede cerrar una comanda ya cerrada	2.

Método modificarProducto

Escenario:

1. Producto distinto de null (por precondición nunca recibiremos uno null)

Condición de entrada	Clases correctas	Clases incorrectas
producto	x = {producto no null} 1.	
nombre	x = {nombre no vacío} 2.	
precioCosto	x = { precioCosto>=0 } 3.	x = {precioCosto<0} 4.
precioVenta	x = {precioVenta>=precioCosto &&	x = {precioVenta<0} 6.

	precioVenta>0} 5.	x = {precioVenta<precioCosto} 7.
stockInicial	x = { stockInicial>=0 } 8.	

Tipo de clase	Valores de entrada	Resultado esperado	Clases de prueba cubiertas
Correcta	(Producto={2, ”Hamburguesa”, 25, 50, 2},”Hamburguesa”, 30, 60, 3) Escenario 1	Se modifica correctamente el producto	1. 2. 3. 5. 8.
Incorrecta	(Producto={2, ”Hamburguesa”, 25, 50, 2},”Hamburguesa”, -4, 60, 3) Escenario 1	El precio de costo es menor a cero	4.
Incorrecta	(Producto={2, ”Hamburguesa”, 25, 50, 2},”Hamburguesa”, 30, 10, 3) Escenario 1	El precio de venta es menor al precio de costo	7.
Incorrecta	(Producto={2, ”Hamburguesa”, 25, 50, 2},”Hamburguesa”, 10, -5, 3) Escenario 1	El precio de venta es menor a cero	6.

Gracias al método de caja negra pudimos observar que al probar el código, el mismo jamás lanzará la excepción cuando el precio de venta sea menor a cero por lo que la misma resulta redundante. Posteriormente, nos pusimos a determinar el motivo por el cual esto ocurre y determinamos que se debe a que si la misma es negativa, el precio de costo deberá ser negativo y de menor valor para no entrar a la primera excepción (precioVenta<precioCosto) y en caso de cumplirse terminará entrando siempre a la segunda excepción (precioCosto<0).

Método mostrarEstadisticaMozo

Este método resultó de gran interés ya que el mismo posee estructuras de decisión y además itera colecciones. El mismo nos devuelve un arreglo de cadenas de texto

con información relevante sobre los mozos. Por precondición sabemos que el array de estadísticas no será nulo, por lo que planteamos dos escenarios con dicha estructura vacía y una con los elementos cargados.

Escenario:

1. Array de estadísticas mozo no nulo

Condición de entrada	**Clases correctas**	**Clases incorrectas.**
Estado de array estadisticasMozo	x = { array con datos} 1. x = { array vacío } 2.	

Tipo de clase	Valores de entrada	Resultado esperado	Clases de prueba cubiertas
Correcta	Array vacio Escenario 1	Se devuelve array vacío	1.
Correcta	Array con mozos cargados Escenario 1	Se devuelve array de string con respuesta del mozo con mayor y menor volumen	2.

Método aplicarPromocionesTemporales

Decidimos utilizar este método, ya que el mismo corresponde a la clase Factura, está relacionado con el cálculo de la cuenta y nos pareció relevante mostrar un método que no esté incluido en la clase Cervecería. Además este método es crucial, ya que estará relacionado con el cálculo del total.

Escenario:

1. Array de promociones temporales no nulo

Condición de entrada	**Clases correctas**	**Clases incorrectas.**
Estado de array promocionesTemporales	x = { array con datos} 1. x = { array vacío } 2.	

Tipo de clase	Valores de entrada	Resultado esperado	Clases de prueba cubiertas
Correcta	Coincide el día de la semana, la promoción está activa y el método de pago también. Array de promociones no vacío Escenario 1	Se aplica el descuento correspondiente	1.
Correcta	Array de promociones vacío Escenario 1	No se aplican descuentos	2.

Al realizar los test de caja negra pudimos observar que a la hora de aplicar una promoción temporal el resultado arrojado no era el esperado. A la hora de devolver el total a pagar lo hacía de manera errónea. Por ejemplo, si nuestro subtotal era de $100 y debíamos aplicar un 25% de descuento por pago en efectivo, la misma nos devolvió un total de $25 en vez de los $75 esperados. Es así, como el método de caja negra nos permitió encontrar un error en el cálculo y permitió darnos una idea de cual podría llegar a ser el problema.

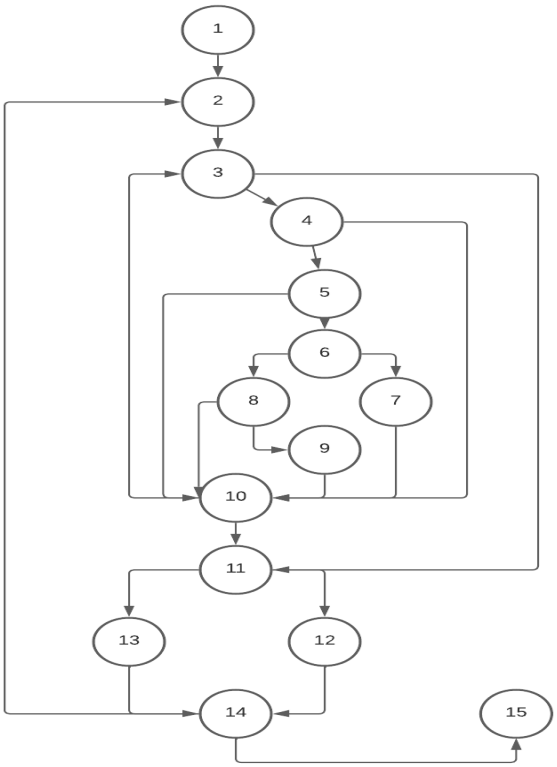
Finalmente luego de haber realizado los tests se hizo la cobertura de cada método y se pudo ver que todos ellos a excepción de uno, todos tenían una cobertura del 100%. El testing de caja negra se vio facilitado ya que se contaba con la documentación de la mayoría de los métodos y clases lo que permitió el fácil entendimiento del código.

Test de caja blanca

A diferencia de los test de caja negra, los de caja blanca se centran en los detalles procedimentales del software por lo que el método está fuertemente ligado al código fuente. Las mismas están enfocadas en las funciones internas del elemento a analizar y se realizan sobre un módulo en específico, esto es, sobre un subsistema en concreto.

A la hora de decidir sobre qué método aplicar, tuvimos en cuenta la utilización de uno que tenga la mayor cantidad de caminos, con estructuras de decisión y bucles de tipo for y while. Por este motivo fue que decidimos testear el método *aplicarPromocionesProductos*, el cual había sido testeado previamente en caja negra y no cumplía el 100% de la cobertura. El primer paso fue realizar un grafo con

todos los caminos posibles para tener una idea más clara previo a la codificación. Obteniendo el siguiente grafo.



Luego de analizar el grafo, se puede determinar que hay 7 nodos condición lo que significa que la complejidad ciclomática es 8 . Para poder recorrer todo el grafo solamente nos basta con la existencia de los siguientes tres caminos, buscando abarcar cada uno de estos caminos creamos un test para cada camino.

Camino 1:

1-2-3-4-5-6-7-10-11-12-14-15

Este camino se recorre en el caso de que uno de los pedidos de la comanda tenga una promoción de producto tipo 2x1 que deberá ser aplicada en el cálculo del total de la factura. El test fue realizado en la clase aplicarPromocionesProductosTest1 con el escenario necesario cargado previamente en la clase EscenarioAplicarPromocionesProductosTest1.

Camino 2:

1-2-3-4-5-6-8-9-10-11-12-14-15

Este camino se recorre en el caso de que uno de los pedidos de la comanda tenga una promoción de producto tipo aplica descuento por cantidad mínima y el número de productos de ese tipo sea mayor o igual a la cantidad mínima necesaria para

aplicar ese descuento, el mismo se verá reflejado en el cálculo del total de la factura. El test fue realizado en la clase aplicarPromocionesProductosTest2 con el escenario necesario cargado previamente en la clase EscenarioAplicarPromocionesProductosTest2.

Camino 3:

1-2-3-11-13-14

Este camino se recorre en el caso de no tener ninguna promoción para los productos que se quieren adquirir, por lo que el cálculo del total no se verá afectado por ningún descuento y solamente dependerá del precio y la cantidad de unidades adquiridas. El test fue realizado en la clase aplicarPromocionesProductosTest3 con el escenario que se encuentra dentro de la clase EscenarioAplicarPromocionesProductosTest3.

Casos de prueba:

Escenario 1: Dentro del array de promocionesProductos que se encuentra en la cervecería, hay 2x1 en hamburguesa y está activo. El cliente consume 2 Hamburguesa de 40 pesos y 3 Panchos de 20.

Escenario 2: Dentro del array de promocionesProductos que se encuentra la cervecería, hay descuento por cantidad mínima en hamburguesas, consumiendo 3 quedan cada una 35 pesos. El cliente consume 5 Hamburguesa de 40 pesos y 3 Panchos de 20.

Escenario 3: El array de promocionesProductos se encuentra vacío y el cliente consume 2 Hamburguesas y 5 panchos

Camino	Escenario	Parámetros	Salida Esperada
C1	1	()	100
C2	2	()	235
C3	3	()	180

Finalmente, luego de aplicar las técnicas de Caja Blanca se realizó el test de cobertura sobre el método para verificar que se cumpla el 100% del mismo ya que anteriormente a la hora de realizar el test de caja negra su cobertura no era máxima.

Coberturas:

Cobertura del método del 35%

```
    public void aplicarPromocionesProductos (){
        ArrayList<PromocionProducto> promoProd = Cerveceria.getInstance().getPromocionesProductos();
        boolean respuesta;
        PromocionProducto promo;
        int pos;

        for (Pedido pedido : this.pedidos){
            pos = 0;
            respuesta = false;
            while(!respuesta && (pos < promoProd.size())){
                promo = promoProd.get(pos);
                if (pedido.getProducto().equals(promo.getProducto())) {
                    if (promo.isActiva() && coincideDiaSemana(promo.getDiasPromocion())) {
                        if (promo.isAplicaDosPorUno()) {
                            //aplicar 2x1 y sumar al total
                            this.total += (Math.divideExact(pedido.getCantidad(), 2)
                                    + Math.ceilMod(pedido.getCantidad(), 2))
                                    * pedido.getProducto().getPrecioVenta();
                            respuesta = true;
                        } else if (promo.isAplicaDtoPorCantidad() && pedido.getCantidad() >= promo.getDtoPorCantidad_CantMinima()) {
                            //aplicar Dto por cantidad y sumar al total
                            this.total += promo.getDtoPorCantidad_PrecioUnitario() * pedido.getCantidad();
                            respuesta = true;
                        }
                    }
                }
                pos++;
            }
            if (respuesta) {
                pos--;
                promocionesAplicadas.add(promoProd.get(pos));
            } else {
                this.total += pedido.getProducto().getPrecioVenta() * pedido.getCantidad();
            }
        }
    }

    /**
```

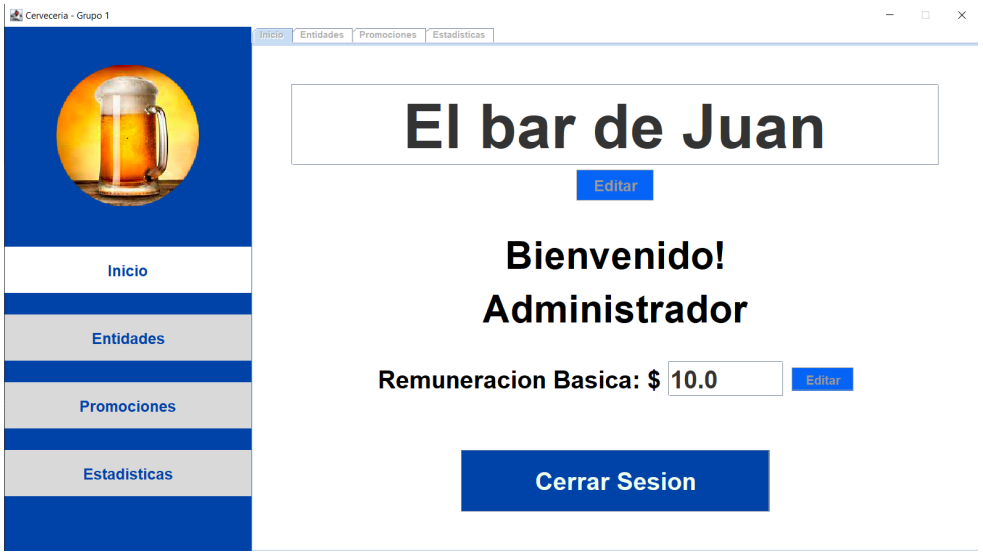
Cobertura del método del 100%

```
    public void aplicarPromocionesProductos (){
        ArrayList<PromocionProducto> promoProd = Cerveceria.getInstance().getPromocionesProductos();
        boolean respuesta;
        PromocionProducto promo;
        int pos;

        for (Pedido pedido : this.pedidos) {
            System.out.println(promoProd.size());
            pos = 0;
            respuesta = false;
            while(!respuesta && (pos < promoProd.size())){
                promo = promoProd.get(pos);
                if (pedido.getProducto().equals(promo.getProducto())) {
                    if (promo.isActiva() && coincideDiaSemana(promo.getDiasPromocion())) {
                        if (promo.isAplicaDosPorUno()) {
                            //aplicar 2x1 y sumar al total
                            this.total += (Math.divideExact(pedido.getCantidad(), 2)
                                    + Math.ceilMod(pedido.getCantidad(), 2))
                                    * pedido.getProducto().getPrecioVenta();
                            respuesta = true;
                        } else if (promo.isAplicaDtoPorCantidad() && pedido.getCantidad() >= promo.getDtoPorCantidad_CantMinima()) {
                            //aplicar Dto por cantidad y sumar al total
                            this.total += promo.getDtoPorCantidad_PrecioUnitario() * pedido.getCantidad();
                            respuesta = true;
                        }
                    }
                }
                pos++;
            }
            if (respuesta) {
                pos--;
                promocionesAplicadas.add(promoProd.get(pos));
            } else {
                this.total += pedido.getProducto().getPrecioVenta() * pedido.getCantidad();
            }
        }
    }
```

Test de GUI

Realizar pruebas sobre las interfaces gráficas es una parte fundamental de nuestro trabajo como testers, si bien es imposible testear totalmente una interfaz podremos simular los comportamientos más comunes para poder ver si posee errores o incongruencias. La ventana elegida para realizar el testing fue la primera que aparece una vez que el administrador inicia sesión.



La misma nos pareció interesante, ya que tiene cajas de texto para modificar el nombre y la remuneración básica acompañadas de un botón para editarla que permanece deshabilitado siempre y cuando no hayan ocurrido cambios en el nombre. Además, el botón *Cerrar Sesion* nos redirige a la ventana de login y por último la ventana cambia de página al presionar los botones Entidades, Promociones y Estadísticas. Al tener tantas funcionalidades para ser probadas que pueden fallar nos pareció una gran elección.

Para poder corroborar que dichas acciones se ejecutan de la manera esperada se programó un robot encargado de la automatización de las mismas. El mismo tendrá los siguientes métodos:

- **testNombreNoModificado():** el mismo no modifica el nombre de la cerveceria y valida si el botón “Editar” permanece deshabilitado.
- **testNombreModificado():** el mismo modifica el nombre de la cervecería y valida si el botón “Editar” permanece habilitado.
- **testCambiaCerrarSesion():** el mismo presiona el botón “Cerrar Sesion” y verifica si cambia a la ventana de login.
- **testCambiaAInicio():** el mismo presiona el botón “Inicio” y verifica si no ocurre un cambio de página.
- **testCambiaAEntidades():** el mismo presiona el botón “Entidades” y verifica si cambia correctamente de página.

- **testCambiaAPromociones():** el mismo presiona el botón “Entidades” y verifica si cambia correctamente de página.
- **testCambiaAEstadisticas():** el mismo presiona el botón “Entidades” y verifica si cambia correctamente de página.

Test de persistencia

La persistencia del sistema está realizada en un archivo binario, persistiendo es necesario leer y escribir la clase Cervecería que es la encargada de llevar las colecciones que almacenan los datos más relevantes del sistema. Para verificar el correcto funcionamiento de la persistencia se tuvieron en cuenta los siguientes casos:

1. La verificación de la creación correcta del archivo
2. La verificación de la des persistencia de una Cervecería Vacía
3. La verificación de la des persistencia de una Cervecería con datos.
4. La verificación del lanzamiento de excepción a la hora de intentar cerrar un archivo inexistente.

Cada uno de los casos se realizó en un test distinto, nuevamente la aplicación de los mismos nos permitió vislumbrar errores en la implementación del código, cuando se testean los casos 2. y 3. Se buscaba demostrar que los elementos DTO eran iguales cuando se cerraba el archivo que cuando se volvía a abrir. Esto se realizó mediante una sentencia assertEquals(), al no estar implementado el método equals dentro de la clase el test fallaba , una vez corregido el resultado de los test fue el esperado llegando a unos resultados apropiados durante los test de Persistencia.

Test de Integración

Las pruebas de integración se enfocan en la interacción entre unidades, suponiendo que cada una fue probada a nivel de unidad. A este nivel se mezclan aspectos estructurales que relacionan las maneras de interactuar de las unidades y también los aspectos típicamente funcionales.

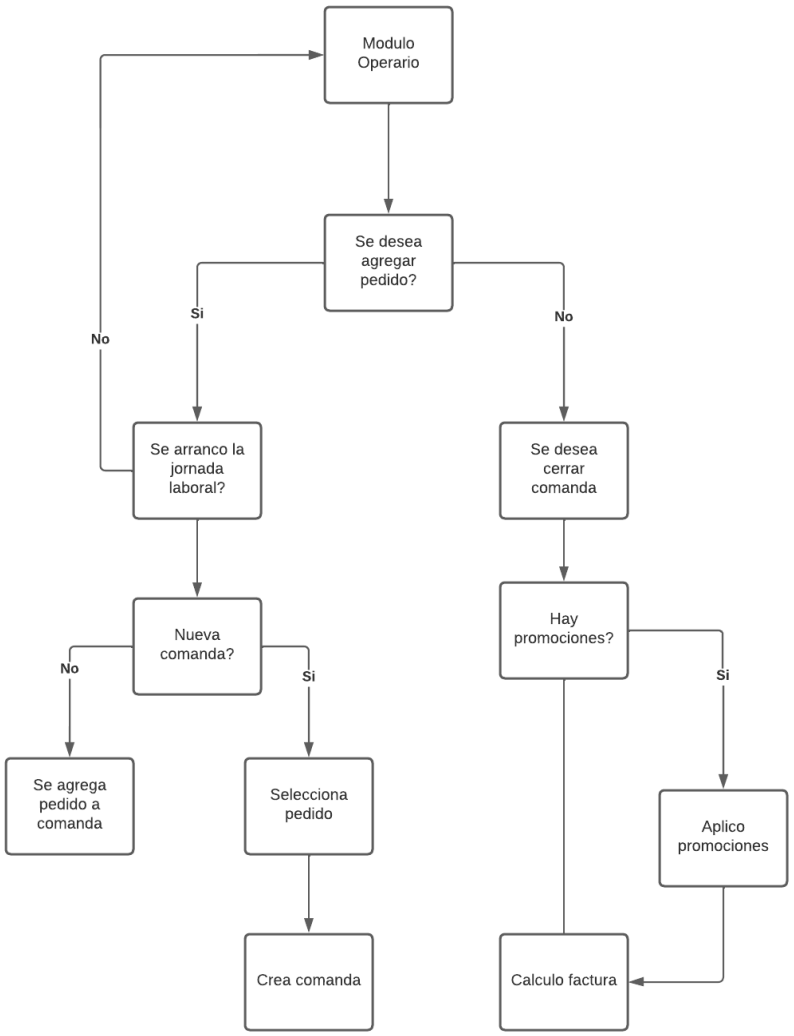
Existen muchos enfoques a las pruebas de integración, destacando los métodos no progresivos y progresivos (bottom-up y top-down). En el caso de software de objetos, cuando se usa algún método derivado de casos de uso, éstos brindan una manera práctica de realizar pruebas de integración, guiándose por los diagramas de colaboración o los diagramas de secuencia asociados con cada caso de uso.

Para la comprensión del problema y el entendimiento de cómo se relacionan las distintas componentes del sistema se realizó un diagrama que simula el flujo principal del mismo, buscando representar las funcionalidades que se utilizaran con mayor frecuencia. Teniendo en cuenta que el sistema está pensado para ser usado

principalmente por los operarios, se pueden reconocer tres casos típicos de uso: la llegada de nuevos clientes al restaurante (crear comanda nueva), agregar pedidos a una comanda y cerrar la comanda.

Se priorizo el flujo de los operarios sobre el administrador ya que van a ser los encargados de utilizar el sistema diariamente, si bien el administrador también tiene acceso a él sus funciones no interactúan con otras componentes por ello fueron testeadas mediante test unitarios, un ejemplo de ellas sería la alta de un mozo.

Diagrama realizado por el método Top-Down:



El diagrama consta de tres ramas principales que representan cada una un caso de uso distinto, para el testeo de cada uno de ellos se simuló un día típico en el restaurante: se cargan los mozos, productos y mesas, se realiza la asignación de mesas y se comienza la jornada laboral. Por cada cada caso de uso se testeo el caso éxito. También se intentó testear los casos de prueba en las que las componentes no existían pero el correcto uso de los asertos y precondiciones nos

avisaban mediante AssertionErrors que no estábamos cumpliendo las precondiciones de los métodos.

Se implementaron dos escenarios: un día típico con promociones y otro sin promociones, en el caso del último al comienzo el test nos daba erróneo. Esto ocurría porque no se consideraba el caso en que el a la hora de arrancar la jornada laboral no había promociones. Mediante la implementación de un test unitario este detalle se había pasado por alto, a la hora de comprender cómo trabajan las distintas componentes en conjunto permitió detectar esta anomalía en el código.

Conclusión:

Tras el análisis, se puede deducir que luego de la utilización de las distintas técnicas de testing sobre el software generó resultados positivos y determinantes sobre la versión final del producto. Cada técnica nos otorga información útil sobre nuestro programa, también permitió percatarnos de detalles que no se pueden descubrir a simple vista.

Caja negra fue la técnica utilizada sobre la mayoría de los métodos del programa priorizando aquellos con una mayor complejidad o que podrían llegar a generar falencias en el código. Por su parte, las pruebas de caja blanca resultaron de gran utilidad en métodos con muchos caminos posibles permitiendo hacer un mejor seguimiento sobre la cobertura de los mismos. Ambos métodos resultaron de gran utilidad para la detección de errores que si bien no eran de gravedad, no permitían el flujo esperado del programa.

Los test de persistencia aseguraron el correcto funcionamiento de los archivos binarios encargados de persistir todo el programa. En un principio, el programa no persistía de la manera esperada pero luego de detectada la falla, logramos que el programa funcione de la manera esperada.

Los tests de integración resultaron de gran ayuda para obtener errores que en los test de unidad no se detectaban por sí solos, era necesario que las distintas componentes trabajarán en conjunto para poder detectarlos. Por su parte, el análisis orientado a objetos nos otorgó una mejor comprensión del problema que ayudó a la hora de la detección de distintos errores.

Cabe destacar que se pudo apreciar resultados positivos aplicando el testing en una etapa tardía del desarrollo del sistema, si se aplicaran técnicas de testing durante etapas tempranas las ventajas serán mayores ya que se ahorraría tiempo y dinero al detectar errores que más adelante podrían generar problemas de mayor complejidad o porque permitirán producir un producto más robusto y de mejor calidad.

Para finalizar, nos gustaría mencionar que el trabajo fue de gran utilidad para reflejar la importancia del testing en el proceso de desarrollo, si bien el mismo en muchas ocasiones resulta tedioso y poco importante, nos pudimos dar cuenta de como simples errores que tienen una pronta solución pueden arruinar el flujo esperado del programa.