Universidad ORT Uruguay

Facultad de Ingeniería

Ingeniería de Software 1
Letra del Obligatorio 2

Santiago Rugnitz (215381)
Nahuel Biladoniga (211138)

Entregado como requisito de la materia Ingeniería de Software 1

25 de noviembre de 2019

Declaraciones de autoría

Nosotros, Santiago Rügnitz (215381) y Nahuel Biladoniga (211138) , declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Ingeniería de Software I ;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Resumen

Documento sobre el proyecto entregado como segundo obligatorio para la materia Ingeniería de Software 1. Presenta las prácticas tecnológicas y de gestión de la ingeniería de software aplicadas para el desarrollo de la misma y como éstas afectaron el resultado final.

Índice general

1. Versionado

1.1. Repositorio utilizado

Para el desarrollo de la aplicación se creó un repositorio[1] en GitHub[2].

El repositorio se organizó con la estructura por defecto de NetBeans[3] y se agregó una carpeta leaflet[4] en la carpeta src para los archivos relacionados con el mapa de los puntos de venta. Además se usó JFoenix[5], una librería para JavaFX[6] que permite diseñar los materiales de los componentes de JavaFX logrando así un amplia mejoría gráfica del programa. La versión utilizada es la 8.0.8. El repositorio se gestionó usando GitKraken[7], una interfaz gráfica que permite hacer commits, merge y otras acciones.

1.2. Criterios de versionado

Para realizar el trabajo distribuido originalmente se crearon dos ramas: master y develop, con el objetivo de realizar los commits a develop y hacer el merge con master cuando se tenga una versión estable. Durante el desarrollo nos vimos en la necesidad de crear una rama más desde develop para el desarrollo de funciones específicas de la interfaz.

En los comentarios de los commits nos aseguramos de que queden claros los cambios aplicados y en caso de haber errores especificar la clase afectada. Además se usó tags en GitKraken para marcar las diferentes versiones del sistema. En las versiones 0.X aún no se habían implementado todas las funciones o no funcionaban correctamente, a partir de la versión 1.0 todas las funciones podían ser realizadas y solo se realizaron pequeñas correcciones de mejoramiento de calidad y correcciones relacionadas con la entrega del jar.

1.3. Resumen del log de versiones

Desde el comienzo del desarrollo decidimos separar tareas: uno se concentró en la interfaz e investigar sobre JavaFX mientras que el otro se centró en el backend. Tras la versión 0.2 prácticamente se termino el backend por lo que ambos integrantes del equipo pasamos a contribuir en el desarrollo de la interfaz, las contribuciones de cada uno son visibles en el historial de commits del repositorio.

Versión 0.1: Se terminó el funcionamiento interno del backend y estructura general del frontend.

Versión 0.2: Se crea el carrito y las funciones principales de la tienda.

Versión 0.3: Se completó el historial y lo relacionado a HTML: la factura y el mapa.

Version 0.4: Se crea la seccion de propuestas. Todas las funcionalidades para el cliente terminadas.

Versión 0.5: Se crea la sección de estadísticas y los métodos en backend necesarios para su funcionamiento.

Versión 0.6: Se agrega la sección de gestión de los artículos de la tienda. Se completaron todas las funcionalidades del administrador.

Versión 1.0: Se agrega el jar al repositorio, se agregan los datos por defecto en el main y se solucionan bugs.

Versión 1.1: Se realiza el testing en la versión anterior y se hacen las correcciones correspondientes.

Versión 1.2: Se solucionan problemas de rutas dentro del jar.

2. Codificación

2.1. Estándar de codificación

Durante el desarrollo se siguió el estándar de codificación de Java visto en el curso:

- Clases comienzan en mayúscula
- Paquetes y métodos comienzan en minúscula
- Uso de camelCase para diferenciar palabras
- Constantes literales todo en mayúsculas (no aplica a nuestro caso)
- Llaves estilo Kernighan y Ritchie

Ejemplo del estilo Kernighan y Ritchie: la llave se abre en la misma linea que la estructura de control (en este caso el if) y se cierra al mismo nivel de identación que la estructura en una linea separada. En Java se acostumbra seguir este estilo también para llaves de clases y métodos.

```
public int cantVentasHoy() {
    int ret = 0;
    for (Venta venta : ventas) {
        if (venta.getFecha().equals(LocalDate.now())) {
            ret++;
        }
    }
    return ret;
}
```

2.2. Pruebas unitarias

Para las pruebas unitarias se utilizo la herramienta JUnit[8]. Se buscó probar todos los métodos del backend y en caso de ser necesario usar varios test para un mismo método para cubrir todos los caminos posibles. Para los métodos de acceso a listas no se creo un test específico ya que se prueba indirectamente con los test de los métodos que las manejan. Una vez implementadas las pruebas, se verificó usando la herramienta JacoCoverage[9] la cobertura de las mismas.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main.java		0%		0%	5	5	56	56	3	3	1	1
Articulo.java		87%		75%	8	39	5	77	6	35	0	3
Compra.java		70%		50%	8	18	9	37	2	12	0	1
Propuesta.java		78%		64%	7	21	8	45	2	14	0	1
Venta.java		90%		77%	7	22	7	57	2	11	0	1
Envase.java		91%		75%	5	23	6	48	2	17	0	1
Sistema.java		100%		91%	5	71	0	135	0	43	0	1
Total	665 of 2.129	69%	30 of 128	77%	45	199	91	455	17	135	1	9

Figura 2.1: Tabla generada por JacoCoverage

Ya que no se probaron los métodos equals() y toString() no se alcanzó el 100 % de cobertura en todas las clases, pero si se comprobó individualmente si todos los demás métodos de cada clase eran debidamente probados.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
equals(Object)		79%		62%	3	5	3	10	0	1
hashCode()		0%		n/a	1	1	2	2	1	1
toString()		0%		n/a	1	1	1	1	1	1
admiteElTipo(Articulo.Tipo)		100%		100%	0	3	0	4	0	1
Envase()		100%		n/a	0	1	0	7	0	1
Envase(String, int, Articulo.Tipo[], int)		100%		n/a	0	1	0	7	0	1
aumentarUso(int)		100%		n/a	0	1	0	2	0	1
setNombre(String)		100%		n/a	0	1	0	2	0	1
setId(int)		100%		n/a	0	1	0	2	0	1
setVecesUsado(int)		100%		n/a	0	1	0	2	0	1
setTipos(Articulo.Tipo[])		100%		n/a	0	1	0	2	0	1
setCosteProduccion(int)		100%		n/a	0	1	0	2	0	1
getNombre()		100%		n/a	0	1	0	1	0	1
getId()		100%		n/a	0	1	0	1	0	1
getVecesUsado()		100%		n/a	0	1	0	1	0	1
getTipos()		100%		n/a	0	1	0	1	0	1
getCosteProduccion()		100%		n/a	0	1	0	1	0	1
Total	13 of 138	91%	3 of 12	75%	5	23	6	48	2	17

Figura 2.2: Tabla de cobertura para la clase Envase

2.3. Análisis del código

A continuación se explicarán los métodos y decisiones tomadas con el objetivo de maximizar la calidad del código. Durante la escritura del código fue imperante mantener la separación entre frontend y backend, manteniendo la lógica del sistema y el manejo de datos en backend y frontend solo con acceso a la clase Sistema. Asi mismo, dentro del backend se tuvo en cuenta el ocultamiento de información y accesibilidad mínima, incluso dentro de la misma clase. En el siguiente ejemplo se ve como el constructor no accede directamente al atributo sino que asigna su valor por medio del set.

```
public Envase() {
    this.setNombre("");
    this.setId(0);
    this.setTipos(new Articulo.Tipo[0]);
    this.setCosteProduccion(0);
    this.setVecesUsado(0);
```

```
}

public Envase(String nombre, int id, Articulo.Tipo[] tipos, int coste
    this.setNombre(nombre);
    this.setId(id);
    this.setVecesUsado(0);
    this.setTipos(tipos);
    this.setCosteProduccion(costeProduccion);
}
```

Para facilitar la lectura del código y las pruebas de los métodos, siempre que fuera necesario se usaron métodos auxiliares para no dar muchas responsabilidades a un solo método y alargarlo innecesariamente.

```
public String registrarVenta() {
    ventas.add(carrito);
    ventasCliente.add(carrito);
    String ret = carrito.generarTicketDGI();
    for (Compra compra : carrito.getCompras()) {
        this.actualizarBeneficio(compra.getEnvase(),
        compra.getCantidad());
        compra.getArticulo().aumentarUso(compra.getCantidad());
        compra.getEnvase().aumentarUso(compra.getCantidad());
    }
    carrito = new Venta();
    actualizarListas();
    return ret;
}

/**
 * Reordena las listas segun el orden predeterminado
 */
private void actualizarListas() {
    envases.sort((Envase t, Envase t1) ->
    t.getVecesUsado() - t1.getVecesUsado());
    articulos.sort((Articulo a, Articulo a1) ->
    a.getVecesComprado() - a1.getVecesComprado());
    ventas.sort((Venta v, Venta v1) -> v.getFecha().compareTo(v1.getF
    ventasCliente.sort((Venta v, Venta v1) -> v.getFecha().compareTo(

}
```

Todas las clases de backend cuentan con un constructor por defecto incluso si no es utilizado y todos los atributos son privados y cuentan con sus respectivos métodos get y set. Dichos métodos son públicos a menos que por la naturaleza del atributo se requiera que no se pueda acceder/modificar desde otra clase.

```
public class Envase {

    private String nombre;
    private int id;
    private int vecesUsado;
    private Articulo.Tipo[] tipos;
    private int costeProduccion;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getId() {
        return id;
    }

    private void setId(int id) {
        this.id = id;
    }
}
```

Una vez terminado el código se realizaron varias lecturas buscando y eliminando variables inutilizadas y métodos que no eran llamados. Además se cambiaron algunos nombres de métodos y variables para que sean mas representativos de la función que tienen. Por último, se utilizó la herramienta FindBugs[10] para ver mejoras que se podrían realizar al código. Permitió mejorar el uso de memoria ya que detectó instancias de objetos que no eran utilizadas y no habían sido detectadas en la lecturas realizadas anteriormente. Además, sugirió el uso de un Stringbuffer/StringBuilder en lugar de concatenar Strings dentro del loop que genera el código html del ticket electrónico, lo cual mejora considerablemente el rendimiento. Al concatenar el String usando += este se transforma en un StringBuilder, se anexa el String y luego se vuelve a transformar en un String y este proceso se repite cada iteración. Es mucho más eficiente crear un StringBuilder antes del loop, usar append() y al final concatenar usando el toString().

Method concatenates strings using + in a loop

The method seems to be building a String using concatenation in a loop. In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration.

Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.

For example:

```
// This is bad
String s = "";
for (int i = 0; i < field.length; ++i) {
  s = s + field[i];
}

// This is better
StringBuffer buf = new StringBuffer();
for (int i = 0; i < field.length; ++i) {
  buf.append(field[i]);
}
String s = buf.toString();
```

Figura 2.3: Una de las sugerencias dadas por FindBugs

3. Interfaz de usuario y usabilidad

3.1. Criterios de interfaz de usuario

Para determinar la usabilidad de la interfaz es necesario considerar la eficacia, la eficiencia y la satisfacción del usuario. Es decir, el usuario debe poder lograr el objetivo deseado, en poco tiempo y esfuerzo y que su experiencia sea positiva. A continuación se explican los criterios con los que se construyó la interfaz, teniendo en cuenta los parámetros mencionados anteriormente.

3.1.1. Estilos de interacción

Para todas las secciones del sistema se optó por la manipulación directa usando botones con correspondencia con el mundo real como el carrito. Para la navegación entre secciones se usa la selección a partir de una lista de ítems. Para el caso del administrador también se puede interactuar con el sistema mediante la entrada de datos en formularios (para la gestión de datos de los artículos).

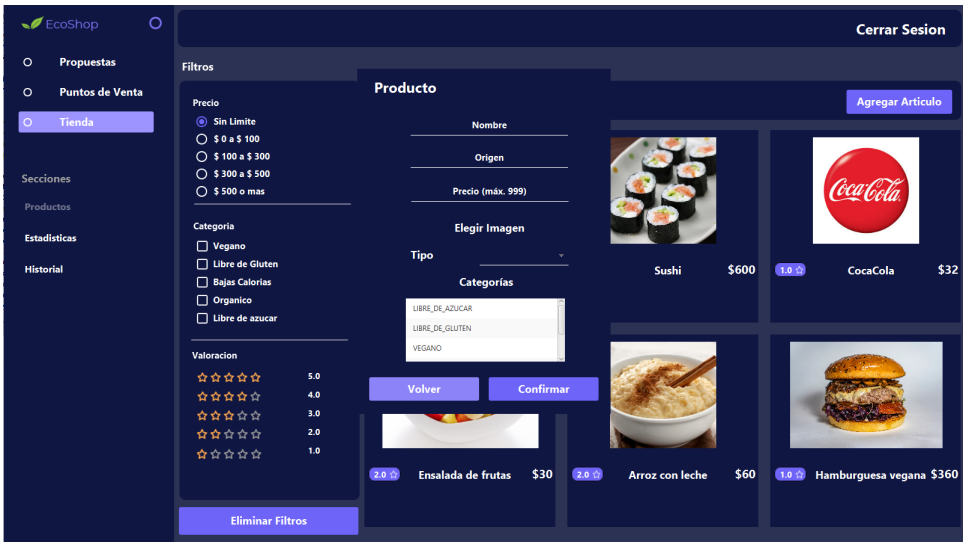

Figura 3.1: Formulario para agregar productos

3.1.2. Tolerancia a errores

Permitirle al usuario la libertad de navegar dentro de la aplicación sin "miedo.[a] realizar una acción irreversible es crucial. La aplicación fue diseñada con este prin-

cipio en mente, en ejemplo de ello es el caso de que la persona por error elige un producto y luego se arrepiente de su decisión, puede ir a la sección de carrito y eliminar dicho elemento de la lista.

3.1.3. Consistencia

Mantener la consistencia del programa permite al usuario tener una mejor experiencia, por ejemplo tener diferentes términos para realizar la misma tarea (aceptar, continuar, confirmar) puede resultar confuso. Algunos ejemplos de consistencia en nuestra aplicación: para cerrar sesión se usa un botón que siempre se encuentra en la misma posición en todas las ventanas del programa, en caso de regresar a una ventana anterior se mostrara un botón "volver", siempre que se vaya a confirmar una acción sin importar cual sea se usará un botón de color verde y en caso de cancelar se usará el color rojo.

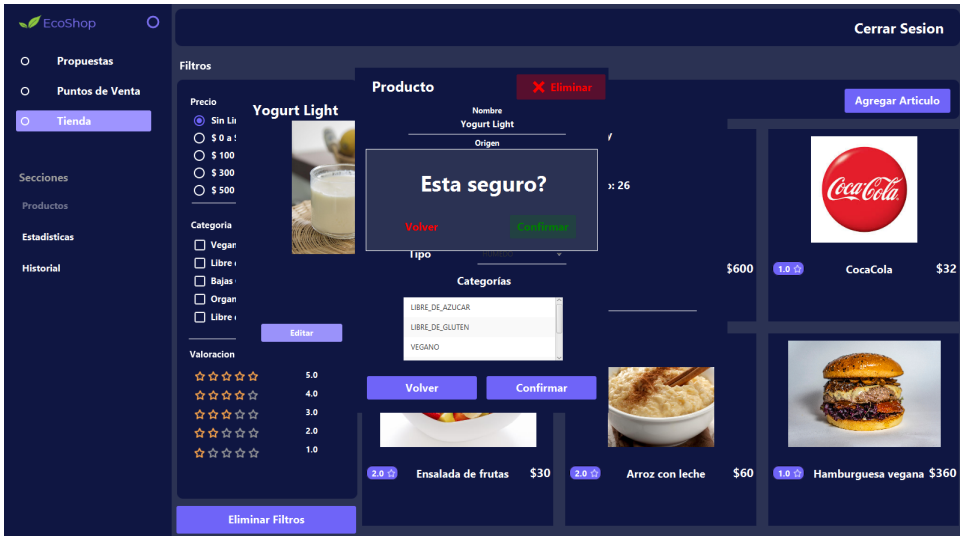

Figura 3.2: Ejemplo de uso de colores en mensaje de error

3.1.4. Llamar la atención del usuario

Lograr la atención del usuario permite que el mismo se sienta mas cómodo con la aplicación y la utilice por mas tiempo, un ejemplo que atrapa a los usuarios es la ventana de tienda con su diseño simple pero con los productos resaltados por un color oscuro frente a un fondo mas claro permitiendo fácilmente identificarlos. Además disponer los elementos en forma de cuadricula con sus respectivas imágenes resulta más atractivo que tener una simple lista con nombres.

3.2. Evaluación de usabilidad

Para evaluar la usabilidad se lo comparó con las heurísticas de Nielsen[11] y el estándar que siguen otras aplicaciones de tiendas online.

3.2.1. Heurísticas de Nielsen

A continuación se explicarán las heurísticas de Nielsen utilizadas y ejemplos de su aplicación.

Visibilidad del estatus del sistema

Es importante dar información sobre el estado del sistema y dar feedback al usuario de las acciones que realiza, debe saber que sus acciones están teniendo un efecto en el sistema. Un ejemplo de esto es al estar en la tienda tener un contador siempre presente de la cantidad de artículos en el carrito que se actualice cada vez que se agrega un nuevo artículo. Esto permite al usuario saber que el articulo se agrega correctamente sin necesidad de abrir el carrito y cambiar de ventana.

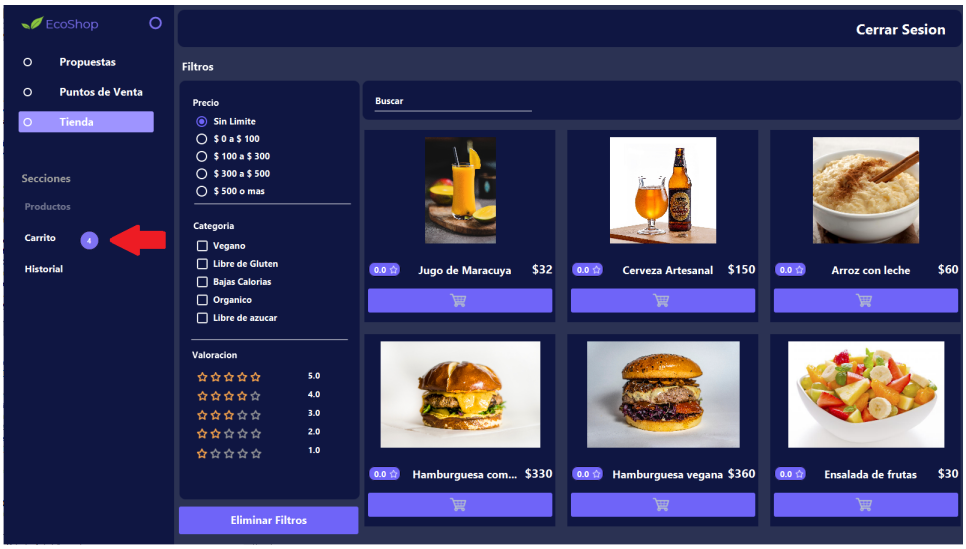

Figura 3.3: Ejemplo de visibilidad del estatus del sistema

Vínculo entre el sistema y el mundo real

Es importante usar símbolos y conceptos relacionados al mundo real que resulten familiares para el usuario. Esto les facilita el entendimiento de la aplicación a los usuarios, especialmente a aquellos que la usan por primera vez. Un ejemplo de esto es usar el símbolo del carrito para guardar los artículos a comprar.

Libertad y control del usuario

Hay que darle al usuario libertad de navegar por la aplicación y explorar todas las opciones y también darle la posibilidad de deshacer cualquier acción accidental. Un ejemplo de esto es mostrar un mensaje de confirmación al presionar el botón de cerrar sesión o al hacer una acción irreversible como borrar un artículo.

Consistencia y estándares

Es importante seguir un estándar y que el formato sea consistente en toda la aplicación. Un ejemplo de esto es que el botón de volver de todas las ventanas emergentes tenga las mismas características: color, texto, etc.

Reconocimiento en vez de memorización

El usuario no debe depender de la memoria para utilizar la aplicación, siempre que es posible todas las opciones deben estar visibles para que el usuario sepa exactamente como realizar la tarea que quiere. En nuestra aplicación, la barra de navegación esta siempre visible permitiendo al usuario saber en todo momento las opciones disponibles y como acceder a ellas.

Diseño minimalista

Es importante no abrumar al usuario con información innecesaria no relacionada con la tarea que quiere desarrollar. Cada información extra compite con lo que realmente le interesa al usuario y le quita visibilidad a lo qué es realmente importante. En nuestra aplicación cada vez que se navega a una sección diferente se oculta la información relacionada a la funcionalidad anterior y se muestra solamente lo estrictamente relacionado con la tarea que el usuario quiere realizar en ese momento.

Ayudar a los usuarios a reconocer, diagnosticar, y recuperarse de los errores

Los mensajes de error deben ser fáciles de comprender para el usuario y ofrecer posibles soluciones al problema, el usuario debe entender qué paso y cómo solucionarlo. En nuestra aplicación al prevenir que el usuario acceda al carrito se muestra un mensaje avisando que el carrito esta vacío, de esta forma el usuario sabe que no es posible abrir un carrito vacío y que la forma de solucionarlo es agregar un artículo.

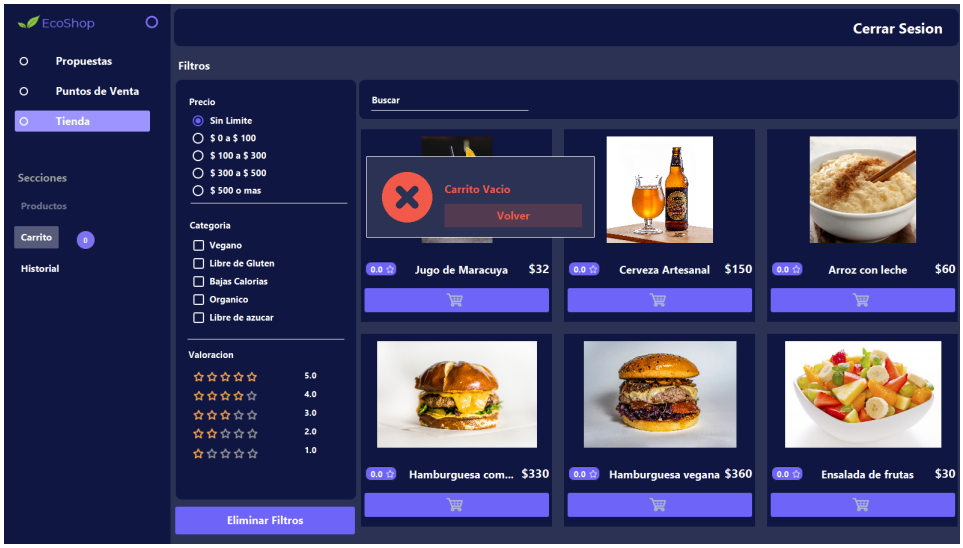

Figura 3.4: Ejemplo de mensaje de error

3.2.2. Estándar en la actualidad

Para verificar que nuestra aplicación sigue el estándar, la comparamos con Mercado Libre[12], Amazon[13], entre otras. Siguiendo estas páginas como ejemplo, llegamos a la conclusión que el estándar en cuanto a la tienda es mostrar los artículos en una cuadricula con los filtros a la izquierda y el buscador arriba. En cuanto al carrito se acostumbra dar la opción de editar individualmente cada articulo del carrito, mostrar cada precio y luego la suma total. Nuestra aplicación ofrece menos funcionalidades que las aplicaciones estudiadas pero se llegó a la conclusión de que hay suficientes características en común para afirmar que sigue el estándar.

4. Pruebas Funcionales

4.1. Técnicas de prueba aplicadas

Se utilizó la prueba exploratoria para comprobar que todas las funcionalidades funcionen correctamente y detectar bugs. Consiste en probar las funcionalidades sin tener datos preestablecidos e improvisar a medida que se avanza. Al casi no haber preparación previa, la eficacia de la prueba depende del razonamiento deductivo y de la habilidad del tester. Se prefirió este estilo de prueba porque es especialmente eficaz cuando el tester conoce el funcionamiento interno de la aplicación y sabe que acciones críticas son las que tienen más probabilidad de generar errores. Además, se aplicó la técnica de prueba de caja negra, la cual consiste en basarse únicamente en la entrada y salida para ver si se cumplen con las especificaciones del software. Dado que no se tiene en cuenta el flujo interno del programa es fundamental elegir correctamente los datos de entrada: usar particiones equivalentes y tener en cuenta valores límites.

4.2. Casos de prueba

Se probará la funcionalidad de agregar productos, cuyo actor es el administrador.

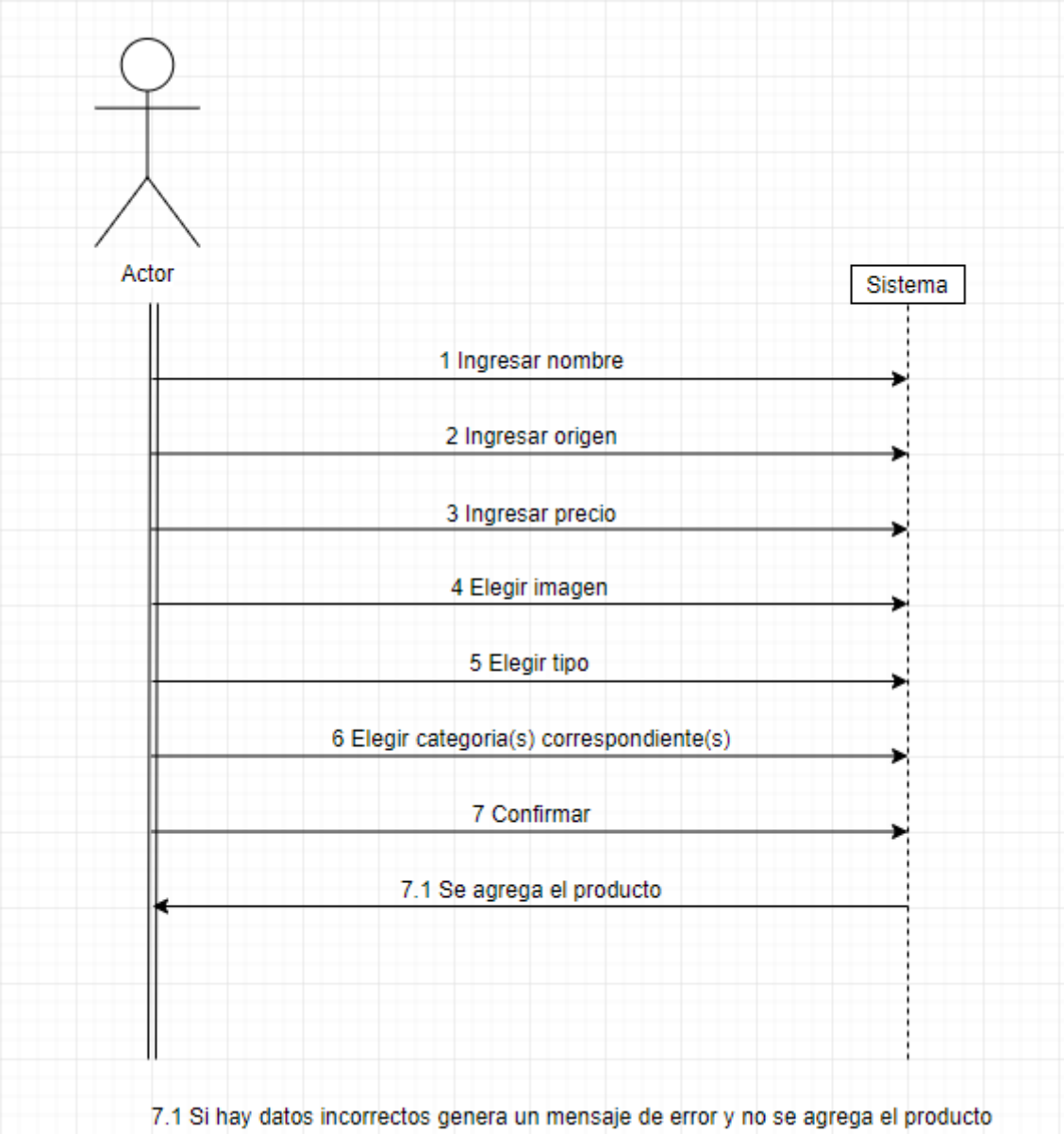

Figura 4.1: Caso de Uso de agregar un producto al sistema

Escenario	Nombre	Curso de comienzo	Curso alternativo
Escenario 1	Agregar producto al sistema	Camino básico	
Escenario 2	Datos incorrectos	Camino básico	CA 7.1

Figura 4.2: Escenarios del caso de uso

Caso de prueba	Escenario	Nombre	Origen	Precio	Imagen	Tipo	Categorías	Resultado esperado
CP 1.1	Escenario 1	V	V	V	V	V	V	Se agrega el producto al sistema
CP 1.2.1	Escenario 2	NV	V	V	V	V	V	Mensaje de Error
CP 1.2.2	Escenario 2	V	NV	V	V	V	V	Mensaje de Error
1.2.3	Escenario 2	V	V	NV	V	V	V	Mensaje de Error
CP 1.2.4	Escenario 2	V	V	V	NV	V	V	Mensaje de Error
CP 1.2.5	Escenario 2	V	V	V	V	NV	V	Mensaje de Error

Figura 4.3: Casos de prueba

Condición	Clases válidas	Clases no válidas
Nombre	String no vacío	No ingresar nada
		Ingresar solo espacios
Origen	String no vacío	No ingresar nada
		Ingresar solo espacios
Precio	1<=precio<=999	No ingresar nada
		precio=0
		precio>999
Imagen	Seleccionar imagen	No seleccionar imagen
Tipo	Seleccionar tipo	No seleccionar tipo
Categorías	Elegir categoría(s)	-
	No elegir categorías	

Figura 4.4: Particiones equivalentes

Caso de prueba	Escenario	Nombre	Origen	Precio	Imagen	Tipo	Categorías	Resultado esperado
CP 1.1.1	Escenario 1	a	a	1	Seleccionada	seco	-	Se agrega el producto al sistema
CP 1.1.2	Escenario 1	a	a	999	Seleccionada	seco	-	Se agrega el producto al sistema
CP 1.1.3	Escenario 1	a	a	5	Seleccionada	seco	vegano	Se agrega el producto al sistema
CP 1.1.4	Escenario 1	a	a	5	Seleccionada	seco	todas	Se agrega el producto al sistema
CP 1.2.1.1	Escenario 2	-	a	5	Seleccionada	seco	-	Mensaje de Error
CP 1.2.1.2	Escenario 2	“ ”	a	5	Seleccionada	seco	-	Mensaje de Error
CP 1.2.2.1	Escenario 2	a	-	5	Seleccionada	seco	-	Mensaje de Error
CP 1.2.2.2	Escenario 2	a	“ ”	5	Seleccionada	seco	-	Mensaje de Error
CP 1.2.3.1	Escenario 2	a	a	0	Seleccionada	seco	-	Mensaje de Error
CP 1.2.3.2	Escenario 2	a	a	1000	Seleccionada	seco	-	Mensaje de Error
CP 1.2.3.3	Escenario 2	a	a	-	Seleccionada	seco	-	Mensaje de Error
CP 1.2.4.1	Escenario 2	a	a	5	No seleccionada	seco	-	Mensaje de Error
CP 1.2.5.1	Escenario 2	a	a	5	Seleccionada	-	-	Mensaje de Error

Figura 4.5: Datos de prueba

Un - indica que no se ingresaron datos

4.3. Sesiones de ejecución de pruebas

4.3.1. Pruebas exploratívas

Pruebas preliminares

Fecha:21/11
Versión: 0.6
Tester: Santiago Rugnitz y Nahuel Biladoniga
Objetivo: Encontrar errores al usar la tienda: agregar elementos al carrito, usar filtros etc.
Resultados: Se encontró el defecto del filtro de la tienda y los relacionados

con el carrito descritos en la sección 5.2. Además, se llegó a la conclusión de que era necesario agregar algún tipo de prevención de errores al sistema para acciones irreversibles como confirmar la compra y cerrar sesión.

Prueba final
Fecha:24/11
Versión: 1.2
Tester: Santiago Rugnitz
Objetivo: Verificar que todas las funcionalidades se pueden realizar correctamente.
Resultados: Se logró realizar todas las tareas que ofrece la aplicación.

4.3.2. Junit

Fecha: 23/11
Versión: 1.0
Tester: Santiago Rugnitz
Objetivo: Cubrir todas las clases del backend.
Resultado: Se comprobó el correcto funcionamiento de todos los métodos probados. En algunos casos fue necesario hacer correcciones. Ej: la lista de articulos se ordenaba de menor a mayor y debería ordenarse de mayor a menor.

4.3.3. Prueba de caja negra

Fecha:24/11
Versión: 1.1
Tester: Santiago Rugnitz
Objetivo: Probar la funcionalidad de agregar artículos al sistema
Resultado: Se obtuvo el resultado esperado en todos los casos probados. No se encontraron defectos.

5. Reporte de defectos

5.1. Definición de categorías de defectos

A continuación se detallará el criterio de clasificación de defectos utilizado.

- **Nivel 1:** Grave, aquellos defectos que comprometen seriamente el funcionamiento de la aplicación al punto de inutilizar varias funcionalidades o incluso el sistema entero.
- **Nivel 2:** Medio, aquellos defectos que en caso de ocurrir inutilizan una funcionalidad del sistema pero no perjudica al resto del sistema.
- **Nivel 3:** Leve, aquellos defectos que no afectan en gran medida la experiencia del usuario ya sea por ser difíciles de reproducir o por no tener un gran impacto en el sistema.

5.2. Defectos encontrados por iteración

5.2.1. Filtro de Tienda

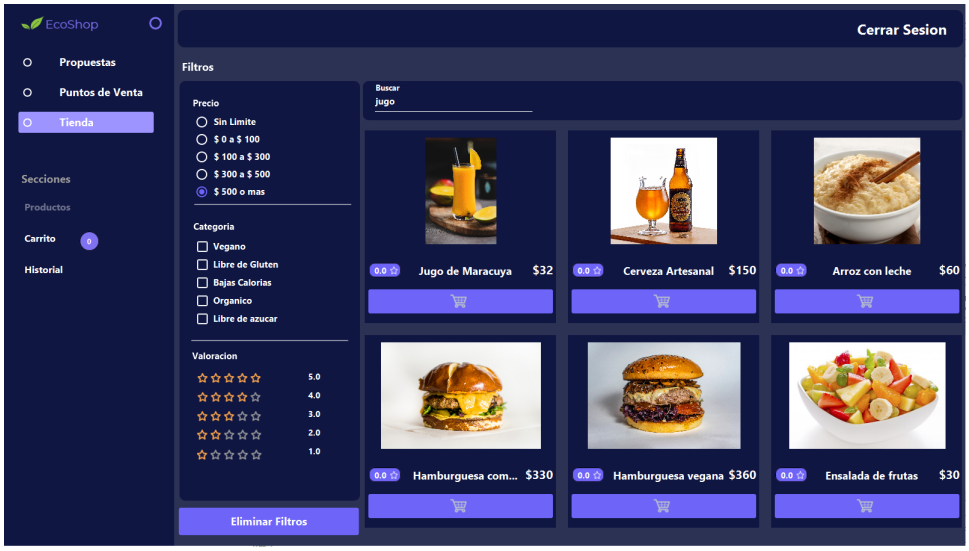

Figura 5.1: Error del filtro de la tienda

Descripción: Al seleccionar la opción de eliminar los filtros, la lista de artículos se reinicia pero los filtros siguen marcados.

Nivel 3: Resulta en una molestia para el usuario pero basta con interactuar con los filtros para que vuelva a funcionar correctamente.

Solución: Devolver a los valores por defecto a los checklist al presionar el botón.

Estado: Solucionado.

5.2.2. Prevención de errores

Descripción: Los botones de cerrar sesión no dan la posibilidad de deshacer la acción, esto puede llevar a que el usuario pierda toda la información de la sesión por error.

Nivel 3: La falta de estos controles no perjudica ninguna funcionalidad del sistema pero si a la satisfacción del usuario.

Solución: Agregar Popups de confirmación antes de puntos sin retorno.

Estado: Solucionado.

5.2.3. Cantidad negativa de artículos

Figura 5.2: Error de cantidad negativa de artículos

Descripción: Una vez agregado el artículo al carrito es posible bajar la cantidad de artículos hasta llegar a un número negativo.

Nivel 1: Si el usuario realiza la compra impacta ambos tipos de historiales y las estadísticas.

Solución: No permitir al usuario disminuir la cantidad si la cantidad actual es 1.

Estado: Solucionado.

5.2.4. Duplicados en Carrito

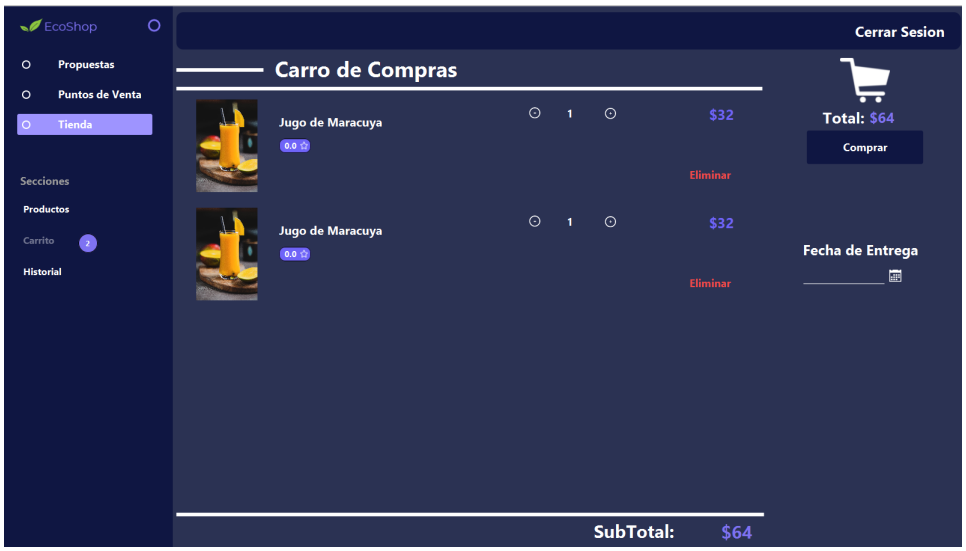

Figura 5.3: Error de artículos repetidos en el carrito

Descripción: Al agregar múltiples veces un mismo articulo al carrito este es considerado como un articulo nuevo.

Nivel 2: Si se confirma la compra, la factura y la venta serían incorrectas pero las estadísticas no se ven afectadas.

Solución: Verificar en backend si el carrito ya contenía el articulo y si es el caso agregar a la cantidad en vez de crear una nueva instancia.

Estado: Solucionado.

5.2.5. Imagen en Historial

Figura 5.4: Error al mostrar los artículos en el historial

Descripción: No se carga la imagen de los artículos en el historial, en su lugar se carga una imagen placeholder.

Nivel 3: A excepción de la imagen, todos los datos del historial funcionan correctamente por lo que no causa un impacto considerable.

Solución: Sustituir la imagen por defecto por la imagen asociada al artículo.

Estado: Solucionado.

5.2.6. Se importa Image en backend

Descripción: En las clases Propuesta y Articulo hay atributos Image que dependen de javafx.

Nivel 3: Imposibilitaría un posible cambio a una interfaz que no depende de javafx pero no afecta el funcionamiento de la implementación actual.

Solución: Cambiar los atributos por path de tipo String o archivos de tipo File. Además, cambiar todos los métodos de backend y frontend que interactúan con estos atributos.

5.2.7. Renderizado del mapa

Descripción: El mapa de Leaflet no se renderiza correctamente en las computadoras del laboratorio.

Nivel 3: Es un error que solo ocurre en casos aislados y el sistema ofrece la alternativa de leer las direcciones de los puntos de venta por lo que el usuario aún puede tener acceso a la información que buscaba.

Solución:

Estado: No solucionado.

5.3. Estado de calidad global

Se solucionaron todos los errores visibles para el usuario encontrados durante las sesiones de ejecución de pruebas. Esto no garantiza que no existan más defectos, pero si se comprobó que todas las funcionalidades puedan ser realizadas y el .jar funcione correctamente con Java 8. Como se mencionó en las secciones 5.2.6 y 5.2.7: el backend depende de que se utilice javafx y el mapa no funciona correctamente en todos los pc, estos defectos no fueron solucionado.

6. Reflexión

Como primer acercamiento a las metodologías de proyectos de desarrollo de software, hemos adquirido experiencia en los distintos campos, desarrollo, testing, entre otros. Sobre las tecnologías utilizadas, JavaFX no conocida por nosotros previamente nos supuso un camino dificultoso pero satisfactorio, dado que permitió tener un resultado superior a usar Java Swing. Administrar los cambios usando la tecnología Git se nos hizo bastante natural por su visualización de árbol, permitiendo trabajar en distintas ramas y poder solucionar los errores fácilmente. Junto al desarrollo y al control de versiones, se encuentra el testing. Fue nuestro primer acercamiento al testing, realizando pruebas unitarias y funcionales las cuales nos permitieron llevar un estándar de como estaba funcionando el sistema. Además fue una oportunidad para investigar sobre herramientas externas como Leaflet y FindBugs que pueden llegar a ser útiles en futuros proyectos.

Bibliografía

[1] Repositorio en github. [Online]. Available: https://github.com/santiagorugnitz/Rugnitz-Biladoniga

[2] GitHub. [Online]. Available: https://github.com/about

[3] NetBeans. [Online]. Available: https://github.com/about

[4] Leaflet. [Online]. Available: https://leafletjs.com/

[5] JFoenix. [Online]. Available: http://www.jfoenix.com/

[6] JavaFX. [Online]. Available: https://openjfx.io/

[7] GitKraken. [Online]. Available: https://www.gitkraken.com/about

[8] JUnit. [Online]. Available: https://junit.org/junit5/

[9] JacoCoverage. [Online]. Available: http://plugins.netbeans.org/plugin/48570/tikione-jacocoverage

[10] FindBugs. [Online]. Available: http://findbugs.sourceforge.net/

[11] J. Nielsen. (1994) 10 Usability Heuristics for User Interface Design. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/

[12] MercadoLibre. [Online]. Available: https://www.mercadolibre.com.uy/

[13] Amazon. [Online]. Available: https://www.amazon.com/

[14] DGI. (2019) Dgi. [Online]. Available: https://bit.ly/35mBDn3