Estructura de
Datos y
Algoritmos

Poli Eats

EDA

Grupo 1

Betancourt Alison

Huilca Fernando

Quisilema Mateo

Simbaña Mateo

ESCUELA POLITÉCNICA NACIONAL

CARRERA DE INGENIERÍA DE SOFTWARE

Estructura de Datos y Algoritmos I (ICCD343)

PROYECTO FINAL

POLI BAR

Integrantes:

Betancourt Herrera Alison Lizeth

Huilca Villagómez Fernando Eliceo

Quisilema Flores Mateo Juan

Simbaña Guarnizo Mateo Nicolás

PROFESORA: Dra. Mayra CARRION

FECHA DE ENTREGA: 13 de agosto de 2024

INFORME PROYECTO FINAL

A. OBJETIVOS

- Desarrollar un programa que simule el funcionamiento de un bar dentro de la facultad de ingeniería de sistemas, utilizando las estructuras de datos y algoritmos aprendidos durante el curso, para gestionar eficientemente las operaciones diarias del bar.
- Diseñar un sistema de colas para gestionar la atención a los clientes de manera ordenada y justa e implementar una pila para manejar las órdenes de los clientes en preparación y entrega rápida.
- Crear una parte en el programa que analice las ventas para identificar los productos más rentables y utilizar algoritmos de clasificación para priorizar los productos ofrecidos, basándose en su demanda.

B. PLANTEAMIENTO DEL PROBLEMA

El administrador del bar, ubicado estratégicamente dentro de la Facultad de Ingeniería de Sistemas, se enfrenta a varios desafíos operativos que impactan directamente la satisfacción de los estudiantes, su principal grupo de clientes. Durante las horas de alta concurrencia, las expectativas de los estudiantes respecto a la calidad y rapidez del servicio son elevadas. Sin embargo, la gran cantidad de pedidos puede resultar en errores operativos, como la entrega incompleta de órdenes o la asignación incorrecta de productos, lo que genera insatisfacción y posibles pérdidas económicas.
Ante estos problemas, surge la necesidad de implementar un sistema integral de gestión para el bar, diseñado específicamente para optimizar el flujo de trabajo en la preparación y entrega de alimentos y bebidas. Este gestor no solo debe simular de manera realista el entorno del bar, sino que también debe incorporar funcionalidades avanzadas que permitan:

- Optimizar el proceso de entrega y la fiabilidad del pedido.
- Mantener un correcto seguimiento del inventario del bar, mediante las categorías de los productos que se tenga.

C. ESTRUCTURAS DE DATOS

Las estructuras de datos se presentan de manera genérica dentro de la implementación del proyecto.

Listas simples

La lista es un tipo de estructura lineal y dinámica de datos, en esta se efectúan las operaciones de recorrido, inserción, borrado y búsqueda, por lo tanto, es necesaria en el contexto del proyecto para almacenar los productos de venta y los ítems de un pedido.

En la clase pedido es utilizada para almacenar y gestionar los ítems, productos que forman parte del pedido, un pedido puede contener múltiples ítems y esta estructura permitirá almacenar y gestionar aquellos de manera eficiente, y, además facilita el recorrido secuencial de estos ítems.

Las listas simples permiten agregar o eliminar elementos de manera dinámica, lo cual es útil en situaciones donde se necesita modificar el tamaño de la lista con frecuencia. En este caso, los productos en el inventario pueden ser añadidos o eliminados fácilmente, ya que se piensa de manera eficiente en el caso de que se añadan más productos, al ser dinámica y no tener número de elementos fijos no existiría inconveniente alguno, además permite acceder a productos específicos dentro del pedido mediante índices.

En la clase AppBarPoliEats es utilizada para gestionar el inventario de productos disponibles, este inventario es esencialmente una colección que puede contener un gran número de elementos, por lo que se podrá realizar las operaciones comunes como agregar, eliminar y acceder a elementos de manera eficiente.

La lista en esta clase se utiliza para agregar nuevos productos al inventario, eliminar productos que ya no estén disponibles, facilitar el acceso a productos específicos y la modificación de estos, y buscar productos por categoría, mediante la realización de operaciones de filtrado.

Colas simples

La cola es una estructura lineal de datos en la que los nuevos elementos se introducen por un extremo y los ya existentes se eliminan por el otro, reciben el nombre de estructuras FIFO, se las puede representar mediante el uso de arreglos o listas, en el contexto del proyecto, como ya se creó la lista simple, se la usó para poder crear la cola simple, en este tipo de estructura se realizan las operaciones básicas de insertar y eliminar.

La naturaleza FIFO de la cola es ideal para manejar los pedidos dentro del entorno del bar, ello asegura que los pedidos se procesen en el orden en que llegan, garantizando que el primero que recibido sea el primer atendido. Además, la inserción de elementos por el final y la eliminación por el frente es un comportamiento que se asemeja con la gestión de pedidos en tiempo real, con esta estructura evitamos la sobrecarga del sistema y se asegura que los pedidos se atiendan de manera continua y en el orden de llegada.

La cola en el código se utiliza:

- Para agregar nuevos pedidos, permitiendo que los pedidos se acumulen en el orden que llegaron.
- Para eliminar el primer pedido de la cola, permitiendo que el siguiente pedido en la cola sea atendido.
- Para visualizar los pedidos actuales, es decir, lo que contiene el pedido, su índice y el nombre del cliente del pedido.

Arreglos

Un arreglo es una secuencia de datos con un número fijo de componentes, todos del mismo tipo, sirven para manejar de forma sencilla y directa conjuntos de datos del mismo tipo, se los puede ver como cajas ordenadas en fila y numeradas. Las operaciones que intervienen en estos son lectura, escritura, asignación, actualización (inserción, eliminación y modificación), ordenación y búsqueda.

En la clase principal (AppBarPoliEats) se utiliza un arreglo para almacenar hasta 500 administradores, el uso de este permite asignar una cantidad fija de espacio en memoria para almacenar los administradores, lo cuál es más eficiente en términos de uso de recursos, además permite un acceso directo a cualquier administrador mediante un índice, y por último, hace que la implementación y mantenimiento del código sean más fáciles.

En el código en esta clase almacena las instancias de Administrador, permitiendo mantener un registro ordenado de hasta 500 usuarios administradores, facilita operaciones como agregar un nuevo administrador, obtener el nombre de un administrador e inicializar los administradores con valores por defecto.

CONCEPTOS NUEVOS

- *Patrón Singleton*

El patrón Singleton es un patrón de diseño que garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella. Este patrón es útil cuando se necesita un único objeto que controle el acceso a un recurso compartido, como una base de datos, una configuración de aplicación o un manejador de tareas.

En el código del proyecto, la clase AppBarPoliEats aplica el patrón Singleton al definir un constructor privado y una instancia estática de la propia clase. La instancia estática es utilizada para almacenar la única instancia de la clase. El constructor privado evita que otras clases puedan crear nuevas instancias de AppBarPoliEats, asegurando así que solo exista una instancia de la clase en todo el ciclo de vida de la aplicación.

El método estático getInstance() es el que permite acceder a esta única instancia. Si la instancia aún no ha sido creada (es decir, si instance es null), el método crea una nueva instancia de AppBarPoliEats y la asigna a instance. Si la instancia ya existe, simplemente la devuelve, asegurando que cualquier parte del código que llame a getInstance() obtenga la misma instancia. De esta manera, el patrón Singleton garantiza que AppBarPoliEats sea una única entidad dentro de la aplicación, lo que es crucial cuando se necesita coordinar el acceso a recursos compartidos, como los administradores, productos y pedidos que maneja esta clase.

- *Generalización de la Lista y Cola Simples*

La generalización es un concepto clave en la programación orientada a objetos que se refiere a la creación de clases o métodos genéricos que pueden trabajar con cualquier tipo de datos. En lugar de crear múltiples versiones de una clase o un método para diferentes tipos de datos, se usa un solo código que se puede aplicar de manera flexible a varios tipos. Esto no solo reduce la duplicación de código, sino que también mejora la mantenibilidad y la reutilización.

En la clase ListaSimple<T>, la generalización permite que la lista maneje cualquier tipo de objeto. La <T> actúa como un parámetro de tipo, que se sustituirá por el tipo concreto que se desee utilizar cuando se instancie un objeto de ListaSimple. Por ejemplo, ListaSimple<Integer> crearía una lista de enteros, mientras que ListaSimple<String> crearía una lista de cadenas de texto. Esto proporciona una gran flexibilidad, ya que la

misma clase ListaSimple puede ser reutilizada para diferentes tipos de datos sin necesidad de modificar su implementación interna.

La clase ColaSimple<T> sigue un enfoque similar, utilizando la clase ListaSimple<T> internamente para gestionar los elementos de la cola. Al aplicar la generalización, ColaSimple<T> puede manejar colas de cualquier tipo de objeto, ya que la ListaSimple que la compone también es genérica. Esto significa que la cola puede almacenar y gestionar elementos de cualquier tipo sin necesidad de cambiar el código de la cola misma. Además, ColaSimple hereda todas las operaciones genéricas de ListaSimple, como agregar y eliminar elementos, lo que simplifica su implementación y extiende su funcionalidad de manera flexible.

- *APPI para la fecha*

El término API (Application Programming Interface) se refiere a un conjunto de reglas y protocolos que permiten que diferentes aplicaciones se comuniquen entre sí. Las APIs definen los métodos y las estructuras de datos que los desarrolladores pueden utilizar para interactuar con un software, servicio o dispositivo.

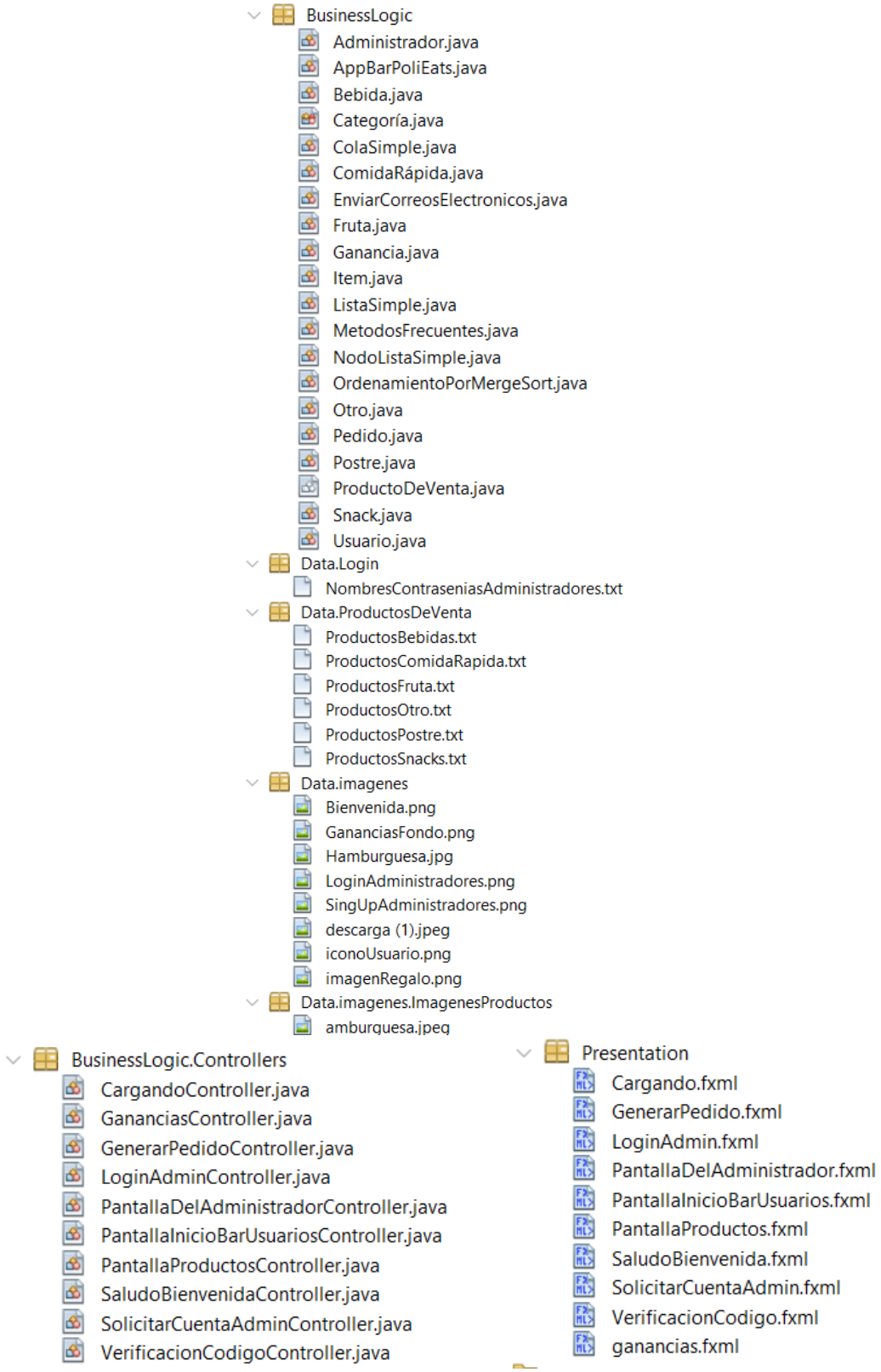
En el contexto de la fecha, una API de Fecha o API de Tiempo es una interfaz que permite a las aplicaciones solicitar y manipular información relacionada con fechas y tiempos. Esto puede incluir la obtención de la fecha y hora actual, conversión entre diferentes zonas horarias, cálculo de diferencias entre fechas, o la manipulación de fechas (como sumar o restar días, semanas, meses, etc.).

D. IMPLEMENTACIÓN

En el proyecto se aplico el patrón de diseño de software modelo-vista-controlador (MVC), el cual se utiliza para separar una aplicación en tres componentes principales, en el contexto de nuestro proyecto como se podrá ver en las siguientes imágenes, establecimos tres packcages principales:

- Business Logic: Encapsula la lógica de negocio, se encarga como tal de las reglas que definen cómo se deben procesar y almacenar los datos.
- Data: Gestiona el almacenamiento y la recuperación de información, en el proyecto son las imágenes y los archivos .txt en los que se almacenan los datos.
- Presentation: Se encarga de la presentación de la información al usuario, proporcionando una interfaz a través de la cual los usuarios interactúan con la aplicación.

El controlador actúa como el intermediario que coordina la interacción entre la vista y el modelo, asegurando que los datos del modelo se presenten correctamente en la vista y que las acciones del usuario se traduzcan en cambios en el modelo.
Esta separación facilita el mantenimiento y la evolución de la aplicación, ya que cada componente puede desarrollarse y modificarse de manera independiente, siempre y cuando se mantenga la interacción correcta entre ellos.



La clase AppBarPoliEats implementa el patrón Singleton, asegurando que solo exista una instancia de esta clase en la aplicación. Esto es útil para gestionar un único punto de acceso a los datos y operaciones relacionadas con la administración de la barra, tales como el manejo de productos, pedidos y administradores. La clase mantiene colecciones de

administradores, productos de venta y pedidos usando arreglos y estructuras de datos personalizadas (ListaSimple y ColaSimple), y proporciona métodos para manipular estos elementos, como agregar, eliminar o modificar productos y pedidos.

La clase también incluye métodos para manejar y consultar información relacionada con las ganancias y los administradores. Ofrece funciones adicionales como la impresión de productos en inventario, filtrado de productos por categoría, y búsqueda y modificación de productos. Además, proporciona mecanismos para gestionar la cola de pedidos, incluyendo la capacidad de agregar, eliminar y consultar pedidos. La implementación del patrón Singleton garantiza que la clase se maneje de manera centralizada, lo cual es crucial para mantener la consistencia en una aplicación que requiere un control único sobre estas operaciones y datos.

```
public class AppBarPoliEats {
    // APLICACIÓN DEL PATRÓN SINGLETON
    private static AppBarPoliEats instance;

    public static final int MAX_ADMINISTRADORES = 500;
    private Administrador[] administradores;
    private ListaSimple<ProductoDeVenta> listSimpleProductos;
    private ColaSimple<Pedido> colaDePedidos;
    private int contadorDeAdministradores;
    private Ganancia[] ganancias;
    private int contadorDeGanancias;

    private AppBarPoliEats() {
        this.administradores = new Administrador[MAX_ADMINISTRADORES];
        this.listSimpleProductos = new ListaSimple<>();
        this.colaDePedidos = new ColaSimple<>();
        this.contadorDeAdministradores = 0;
        this.ganancias = new Ganancia[1000];
        this.contadorDeGanancias = 0;
       // inicializarAdministradores();
    }

    public static AppBarPoliEats getInstance() {
        if (instance == null) {
            instance = new AppBarPoliEats();
        }
        return instance;
    }

    public boolean agregarGanancia(Ganancia ganancia){
        ganancias[contadorDeGanancias++] = ganancia;
        return true;
    }

    private void inicializarAdministradores() {
        for (int i = 0; i < MAX_ADMINISTRADORES; i++) {
            administradores[i] = new Administrador();
        }
    }
```

```
    public boolean agregarAdministrador(Administrador nuevoAdministrador) {
        if (contadorDeAdministradores < MAX_ADMINISTRADORES) {
            administradores[contadorDeAdministradores++] = nuevoAdministrador;
            return true;
        }
        return false;
    }

    public boolean agregarProductoIzquierda(ProductoDeVenta nuevoProducto) {
        return listSimpleProductos.agregarALaIzquierda(nuevoProducto);
    }


    public Administrador[] getAdministradores() {
        return administradores;
    }

    public String getNombreAdministrador(int numAdministrador) {
        return administradores[numAdministrador].getNombre();
    }

    public int getNúmeroDeProductosDelInventario() {
        return listSimpleProductos.getNúmeroDeDatos();
    }

    public ProductoDeVenta getProductoDelInventario(int i) {
        return listSimpleProductos.getDato(i);
    }

    public boolean agregarProductoDerecha(ProductoDeVenta nuevoProducto) {
        return listSimpleProductos.agregarALaDerecha(nuevoProducto);
    }

    public boolean eliminarProductoIzquierda() {
        return listSimpleProductos.eliminarALaIzquierda();
    }

    public boolean eliminarProductoDerecha() {
        return listSimpleProductos.eliminarALaDerecha();
    }
```

```
    public boolean modificarProductoPrecio(int indiceDelProducto, double precio) {
        return listSimpleProductos.getDato(indiceDelProducto).setPrecio(precio);
    }

    public boolean modificarProductoStock(int indiceDelProducto, int cantidadDeStock) {
        return listSimpleProductos.getDato(indiceDelProducto).setStock(cantidadDeStock);
    }

    public void eliminarProductoDelInventario(ProductoDeVenta producto){
        listSimpleProductos.eliminarDato(producto);
    }
    public ProductoDeVenta buscarProductoDelInventario(ProductoDeVenta producto){
        return listSimpleProductos.buscarDato(producto);
    }
    public boolean modificarProductoDelInventario(ProductoDeVenta productoActual, ProductoDeVenta nuevoProducto){
        return listSimpleProductos.modificarDato(productoActual, nuevoProducto);
    }

    public int getContadorDeGanancias() {
        return contadorDeGanancias;
    }

    public Ganancia getGanancia(int indice) {
        return ganancias[indice];
    }
}
```

```java
public boolean eliminarProductoEnElÍndice(int índice) {
    return listSimpleProductos.eliminarEnIndice(índice);
}

public boolean agregarPedido(Pedido nuevoPedido) {
    return colaDePedidos.agregarDato(nuevoPedido);
}

public Pedido getPedido(int i) {
    return colaDePedidos.getDato(i);
}

public int getNúmeroDePedidos() {
    return colaDePedidos.getNumeroDeDatos();
}

public int getContadorAdministradores(){
    return contadorDeAdministradores;
}

public String getNombreDelClienteDelPedido(int índiceDelPedido) {
    return colaDePedidos.getDato(índiceDelPedido).getNombreDelCliente();
}


public void imprimePedido(int índiceDelPedido) {
    System.out.println(colaDePedidos.getDato(índiceDelPedido));
}

public Pedido eliminarPedido() {
    return colaDePedidos.eliminarDato();
}

public void imprimirProductosEnInventario() {
    for (int i = 0; i < listSimpleProductos.getNúmeroDeDatos(); i++) {
        System.out.println(listSimpleProductos.getDato(i));
    }
}
```

```java
//Método para imprimir productos por categoría GENERAL:
public void imprimirProductosDelInventarioPorCategoría(Categoría categoría) {
    for (int i = 0; i < listSimpleProductos.getNúmeroDeDatos(); i++) {
        ProductoDeVenta producto = listSimpleProductos.getDato(i);
        if (producto.getCategoría() == categoría) {
            System.out.println(producto);
        }
    }
}

// Método para devolver el número de productos según alguna categoría
public int getNúmeroDeProductosDelInventarioCategoría(Categoría categoría) {
    int count = 0;
    for (int i = 0; i < listSimpleProductos.getNúmeroDeDatos(); i++) {
        if (listSimpleProductos.getDato(i).getCategoría() == categoría) {
            count++;
        }
    }
    return count;
}

// Método para devolver un arreglo de la lista de productos según la categoría
public ProductoDeVenta[] getProductosDelInventarioCategoría(Categoría categoría) {
    int númeroDeCategoría = getNúmeroDeProductosDelInventarioCategoría(categoría);
    ProductoDeVenta[] auxProductos = new ProductoDeVenta[númeroDeCategoría];
    int contador = 0;
    for (int i = 0; i < listSimpleProductos.getNúmeroDeDatos(); i++) {
        if (listSimpleProductos.getDato(i).getCategoría() == categoría) {
            auxProductos[contador] = listSimpleProductos.getDato(i);
            contador++;
        }
    }
    return auxProductos;
}

public boolean modificarProductoNombre(int índiceDelProducto, String nombre) {
    return listSimpleProductos.getDato(índiceDelProducto).setNombre(nombre);
}
```

Dentro de la clase principal, la que implementa el patrón Singleton, se encuentra un arreglo de administradores, la clase Administrador representa una entidad que almacena información esencial sobre un administrador del sistema. Con tres atributos privados nombre, correo y contraseña. Ofrece dos constructores: uno que inicializa todos los atributos con valores específicos y otro por defecto que asigna null a los atributos, permitiendo la creación de objetos de la clase sin necesidad de proporcionar datos iniciales. Además, la clase proporciona métodos de acceso (getters) para obtener el nombre y la contraseña del administrador, facilitando el acceso a estos datos sin exponer directamente los atributos privados.

```java
public class Administrador {

    private String nombre;
    private String correo;
    private String contraseña;

    public Administrador(String nombre, String correo, String contraseña) {
        this.nombre = nombre;
        this.correo = correo;
        this.contraseña = contraseña;
    }
    public Administrador() {
        this.nombre = null;
        this.correo = null;
        this.contraseña = null;
    }

    public String getNombre() {
        return nombre;
    }

    public String getContraseña() {
        return contraseña;
    }
}
```

Además, en la clase principal se declara una lista simple de tipo Productos de venta, para ello cabe recalcar que se trabajó con una generalización de dos estructuras de datos, las listas y colas simples.

Las clases ListaSimple y NodoListaSimple implementan una estructura de datos de lista enlazada simple utilizando generics, lo que permite que la lista maneje distintos tipos de datos de manera flexible. La clase ListaSimple gestiona una secuencia de nodos donde cada nodo puede contener un dato y una referencia al siguiente nodo. Ofrece métodos para agregar elementos tanto al inicio como al final de la lista, eliminar elementos desde el inicio, el final o en un índice específico, y buscar o modificar elementos en la lista. Este diseño proporciona una forma eficiente de manipular colecciones de datos donde el tamaño puede cambiar dinámicamente.

Por su parte, la clase NodoListaSimple representa cada nodo en la lista enlazada. Esta clase utiliza generics para que el nodo pueda almacenar cualquier tipo de dato. Dispone de métodos para acceder y modificar el dato almacenado (getDato y setDato) así como para manejar la referencia al siguiente nodo (getLiga y setLiga). Los constructores de NodoListaSimple permiten crear nodos con datos y una referencia al siguiente nodo, proporcionando flexibilidad para construir la lista según sea necesario. En conjunto, estas clases proporcionan una implementación básica pero robusta de una lista enlazada simple, adecuada para aplicaciones que requieren operaciones dinámicas sobre una colección de elementos.

```java
public class ListaSimple<T> {
    private NodoListaSimple<T> nodoDeInicio;
    private int contadorDeNodos;

    public ListaSimple() {
        this.nodoDeInicio = null;
        this.contadorDeNodos = 0;
    }

    public boolean agregarALaIzquierda(T dato) {
        NodoListaSimple<T> nuevoNodo = new NodoListaSimple<>();
        nuevoNodo.setDato(dato);

        if (nodoDeInicio == null) {
            nodoDeInicio = nuevoNodo;
        } else {
            nuevoNodo.setLiga(nodoDeInicio);
            nodoDeInicio = nuevoNodo;
        }
        contadorDeNodos++;
        return true;
    }

    public boolean agregarALaDerecha(T dato) {
        NodoListaSimple<T> nuevoNodo = new NodoListaSimple<>();
        nuevoNodo.setDato(dato);

        if (nodoDeInicio == null) {
            nodoDeInicio = nuevoNodo;
        } else {
            NodoListaSimple<T> actual = nodoDeInicio;
            while (actual.getLiga() != null) {
                actual = actual.getLiga();
            }
            actual.setLiga(nuevoNodo);
        }
        contadorDeNodos++;
        return true;
    }
```

```java
public int getNúmeroDeDatos() {
    return contadorDeNodos;
}

public T getDato(int i) {
    if (i < 0 || i >= contadorDeNodos) {
        return null; //TODO: Ver una mejor solución
    }

    NodoListaSimple<T> actual = nodoDeInicio;
    for (int j = 0; j < i; j++) {
        actual = actual.getLiga();
    }

    return actual.getDato();
}

public boolean eliminarALaIzquierda() {
    if (nodoDeInicio == null) {
        return false;
    }

    nodoDeInicio = nodoDeInicio.getLiga();
    contadorDeNodos--;
    return true;
}
```

```java
public boolean eliminarALaDerecha() {
    if (nodoDeInicio == null) {
        return false;
    }

    if (nodoDeInicio.getLiga() == null) {
        nodoDeInicio = null;
    } else {
        NodoListaSimple<T> actual = nodoDeInicio;
        while (actual.getLiga().getLiga() != null) {
            actual = actual.getLiga();
        }
        actual.setLiga(null);
    }
    contadorDeNodos--;
    return true;
}

public boolean eliminarEnIndice(int i) {
    if (i < 0 || i >= contadorDeNodos) {
        return false;
    }

    if (i == 0) {
        return eliminarALaIzquierda();
    }

    NodoListaSimple<T> actual = nodoDeInicio;
    for (int j = 0; j < i - 1; j++) {
        actual = actual.getLiga();
    }
    actual.setLiga(actual.getLiga().getLiga());
    contadorDeNodos--;
    return true;
}
```

```java
public T buscarDato(T dato) {
    NodoListaSimple<T> actual = nodoDeInicio;
    while (actual != null) {
        if (actual.getDato().equals(dato)) {
            return actual.getDato(); // Devuelve el dato si lo encuentra
        }
        actual = actual.getLiga();
    }
    return null; // Devuelve null si no encuentra el dato
}

public boolean eliminarDato(T dato) {
    if (nodoDeInicio == null) {
        return false; // La lista está vacía
    }

    if (nodoDeInicio.getDato().equals(dato)) {
        nodoDeInicio = nodoDeInicio.getLiga();
        contadorDeNodos--;
        return true; // Eliminado el nodo al inicio
    }

    NodoListaSimple<T> actual = nodoDeInicio;
    while (actual.getLiga() != null) {
        if (actual.getLiga().getDato().equals(dato)) {
            actual.setLiga(actual.getLiga().getLiga());
            contadorDeNodos--;
            return true; // Nodo eliminado
        }
        actual = actual.getLiga();
    }
    return false; // El dato no fue encontrado
}
```

```java
public boolean modificarDato(T datoActual, T nuevoDato) {
  NodoListaSimple<T> actual = nodoDeInicio;

  // Recorre la lista buscando el nodo con el dato que se quiere modificar
  while (actual != null) {
      if (actual.getDato().equals(datoActual)) {
          actual.setDato(nuevoDato); // Modifica el dato
          return true; // Indica que la modificación fue exitosa
      }
      actual = actual.getLiga();
  }

  return false; // Devuelve false si no se encontró el dato a modificar
}

}
```

La clase ColaSimple implementa una estructura de datos de cola utilizando una lista enlazada simple como base para almacenar los elementos. Emplea generics para permitir que la cola pueda manejar diversos tipos de datos. En su implementación, la cola se gestiona mediante una instancia de ListaSimple, que se utiliza para agregar y eliminar elementos de manera eficiente. La clase proporciona métodos esenciales para operar con la cola: agregarDato añade un elemento al final de la cola, mientras que eliminarDato elimina el elemento del frente, siguiendo el principio FIFO (First In, First Out) característico de las colas.

El método agregarDato utiliza la funcionalidad de la lista para insertar elementos al final, lo que asegura que el nuevo dato se coloque al final de la cola. Por otro lado, el método eliminarDato elimina el elemento del frente de la cola, accediendo al primer elemento de

la lista y luego eliminándolo de la derecha. La clase también incluye métodos para obtener el número de elementos en la cola (getNumeroDeDatos) y para acceder a elementos en índices específicos (getDato).

```java
public class ColaSimple<T> {
    // Atributos
    private ListaSimple<T> datos;

    public ColaSimple() {
        this.datos = new ListaSimple<>();
    }

    // Método para agregar un pedido al final de la cola
    public boolean agregarDato(T nuevoDato) {
        return datos.agregarALaDerecha(nuevoDato);
    }

    // Método para remover un pedido del frente de la cola
    public T eliminarDato() {
        if (datos.getNúmeroDeDatos() == 0) {
            return null; // La cola está vacía
        }
        T dato = datos.getDato(0);
        datos.eliminarALaDerecha();
        return dato;
    }

    // Método para obtener el número de pedidos en la cola
    public int getNumeroDeDatos() {
        return datos.getNúmeroDeDatos();
    }

    public T getDato(int i) {
        return datos.getDato(i);
    }
}
```

La clase ProductoDeVenta es una clase abstracta que define las características y comportamientos comunes para los productos que se venden. Esta clase encapsula atributos esenciales como nombre, precio, stock, categoría y rutaImagen, proporcionando métodos para acceder y modificar estos valores. El constructor de la clase inicializa estos atributos, mientras que los métodos setNombre, setPrecio, setStock y setRutaImagen permiten actualizar los valores de los atributos correspondientes. Además, el método aplicarDescuento facilita la aplicación de un descuento al precio del producto.

Dado que ProductoDeVenta es una clase abstracta, no se puede instanciar directamente. En su lugar, está diseñada para ser extendida por otras clases que implementen el método abstracto getCategoría, permitiendo una definición específica de la categoría del producto. La clase también sobrescribe los métodos toString, hashCode y equals para proporcionar una representación textual del producto y definir cómo se comparan dos instancias de ProductoDeVenta. El método toString ofrece una representación legible del producto, mientras que equals y hashCode aseguran que la comparación y el almacenamiento del producto en estructuras de datos como conjuntos y mapas sean consistentes y correctos.

```java
public abstract class ProductoDeVenta {
    private String nombre;
    private double precio;
    private int stock;
    private Categoría categoría;
    private String rutaImagen;

    public ProductoDeVenta(String nombre, double precio, int stock, String rutaImagen) {
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
        this.rutaImagen = rutaImagen;
        this.categoría = getCategoría();
    }


    public void aplicarDescuento(double porcentaje) {
        this.precio -= this.precio * (porcentaje / 100);
    }

    @Override
    public String toString() {
        return nombre + " - $" + precio + " Stock : " + stock;
    }

    public boolean setNombre(String nombre) {
        this.nombre = nombre;
        return true;
    }
    public boolean setPrecio(double precio) {
        this.precio = precio;
        return true;
    }

    public boolean setStock(int cantidadDeStock) {
        this.stock = cantidadDeStock;
        return true;
    }
```

```java
public String getNombre() {
    return nombre;
}
public int getStock(){
    return stock;
}

public double getPrecio() {
    return precio;
}
public String getRutaImagen(){
    return rutaImagen ;
}
public void setRutaImagen(String nuevaRuta){
    this.rutaImagen = nuevaRuta;
}

public abstract Categoría getCategoría();

@Override
public int hashCode() {
    int hash = 3;
    return hash;
}
```

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final ProductoDeVenta other = (ProductoDeVenta) obj;
    if (Double.doubleToLongBits(this.precio) != Double.doubleToLongBits(other.precio)) {
        return false;
    }
    if (this.stock != other.stock) {
        return false;
    }
    if (!Objects.equals(this.nombre, other.nombre)) {
        return false;
    }
    return Objects.equals(this.categoría, other.categoría);
}
```

Dentro de la clase ProductoDeVenta se declara un atributo de tipo categoría, la clase Categoría es una enumeración que define un conjunto de constantes relacionadas con las categorías de productos disponibles en el sistema. Las categorías definidas son BEBIDA, POSTRE, SNACK, COMIDA_RÁPIDA, FRUTA y OTRO, cada una representando un tipo específico de producto. La enumeración encapsula una cadena (String) que describe la categoría, permitiendo que cada constante tenga una representación textual asociada.

El constructor privado de la enumeración Categoría inicializa el atributo categoría con el nombre de la categoría proporcionado al crear una instancia. El método getCategoríaString proporciona una forma de obtener la representación textual de la categoría, lo que facilita la conversión de los valores de la enumeración a una cadena que puede ser utilizada en la interfaz de usuario o en reportes. En general, esta enumeración

organiza y clasifica los productos en categorías específicas, proporcionando un enfoque claro y estructurado para manejar diferentes tipos de productos en el sistema.

```
public enum Categoría {
    BEBIDA("Bebida"),
    POSTRE("Postre"),
    SNACK("Snack"),
    COMIDA_RÁPIDA("Comida rápida"),
    FRUTA("Fruta"),
    OTRO("Otro");
    private String categoría;

    Categoría(String categoría) {
    this.categoría = categoría;
    }

    public String getCategoríaString(){
        return categoría;
    }

}
```

En la clase principal también está una cola simple, de la cual ya se indicó su código, y es de tipo Pedido, para así dar referencia a una cola de pedidos, la clase Pedido representa una orden realizado por un cliente en un sistema de venta, gestionando información sobre el cliente, los ítems del pedido y el valor total a pagar. La clase tiene tres atributos principales: nombreDelCliente, que almacena el nombre del cliente que realizó el pedido; items, una lista de Item que representa los productos incluidos en el pedido; y valorTotalAPagar, que calcula el monto total que el cliente debe pagar, sumando el valor de cada ítem en la lista. El constructor inicializa estos atributos y calcula el valor total en base a los ítems proporcionados.

La clase ofrece métodos para acceder y modificar sus atributos, incluyendo getNombreDelCliente, setNombreDelCliente, getItems, y setItems. Además, proporciona métodos para obtener el número de ítems en el pedido (getNúmeroDeItems), acceder a productos específicos dentro de los ítems (getProductos), y recuperar el valor total a pagar (getValorTotalAPagar). El método toString sobrescribe la representación textual del pedido, generando un string que incluye el nombre del cliente, una lista de los ítems del pedido y el valor total a pagar.

```java
public class Pedido {
    private String nombreDelCliente;
    private ListaSimple <Item> items;
    private double valorTotalAPagar;

    public Pedido(String nombreDelCliente, ListaSimple<Item> items) {
        this.nombreDelCliente = nombreDelCliente;
        this.items = items;
        valorTotalAPagar = 0;
        for (int i = 0 ; i < items.getNúmeroDeDatos(); i++){
            this.valorTotalAPagar += items.getDato(i).getValorAPagar();
        }
    }


    public String getNombreDelCliente() {
        return nombreDelCliente;
    }

    public void setNombreDelCliente(String nombreDelCliente) {
        this.nombreDelCliente = nombreDelCliente;
    }

    public ListaSimple<Item> getItems() {
        return items;
    }

    public void setItems(ListaSimple<Item> items) {
        this.items = items;
    }
    public int getNúmeroDeItems() {
    return items.getNúmeroDeDatos();
    }

    public ProductoDeVenta getProductos(int índiceDelProducto) {
        return items.getDato(índiceDelProducto).getProducto();
    }

    public double getValorTotalAPagar(){
        return valorTotalAPagar;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Pedido ")
                .append("nombreDelCliente ").append(nombreDelCliente).append('\'');

        sb.append(" \n");

        for (int i = 0; i < items.getNúmeroDeDatos(); i++) {
            sb.append(items.getDato(i).toString());
            if (i < items.getNúmeroDeDatos() - 1) {
                sb.append(", ");
            }
        }
        sb.append(" \n Valor a pagar = " + getValorTotalAPagar());

        return sb.toString();
    }
}
```

Dentro de la clase Pedido, se declara una lista simple de tipo Item, la clase Item representa una unidad de un producto en un pedido, encapsulando la información sobre el producto y la cantidad solicitada. Cada instancia de Item contiene un objeto de tipo ProductoDeVenta y un entero que indica la cantidad del producto. El constructor de la clase inicializa estos atributos, asegurando que cada Item tenga tanto el producto asociado como la cantidad correspondiente.

La clase ofrece métodos para acceder y modificar estos atributos a través de getProducto, getCantidadDeEsteProducto, setProducto, y setCantidadDeEsteProducto. Además, el método toString proporciona una representación textual de un Item, que incluye el

producto y la cantidad. El método getValorAPagar calcula el valor total que debe pagar el cliente por este ítem, multiplicando el precio del producto por la cantidad.

```java
public class Item {
    private ProductoDeVenta producto;
    private int cantidadDeEsteProducto;

    public Item(ProductoDeVenta producto, int cantidadDeEsteProducto) {
        this.producto = producto;
        this.cantidadDeEsteProducto = cantidadDeEsteProducto;
    }

    public ProductoDeVenta getProducto() {
        return producto;
    }

    public int getCantidadDeEsteProducto() {
        return cantidadDeEsteProducto;
    }

    public void setProducto(ProductoDeVenta producto) {
        this.producto = producto;
    }

    public void setCantidadDeEsteProducto(int cantidadDeEsteProducto) {
        this.cantidadDeEsteProducto = cantidadDeEsteProducto;
    }

    @Override
    public String toString() {
        return "Item{" +
                "producto=" + producto +
                ", cantidadDeEsteProducto=" + cantidadDeEsteProducto +
                '}';
    }

    public double getValorAPagar() {
        return producto.getPrecio() * cantidadDeEsteProducto;
    }
}
```

Finalmente, en la clase principal también se declara un arreglo de tipo ganancia, la clase Ganancia representa el registro de ganancias para un periodo específico, almacenando la fecha y el monto total de las ganancias obtenidas en ese periodo. Incluye dos atributos principales: fecha, que es una instancia de LocalDate que indica cuándo se registró la ganancia, y totalGanancia, un valor de tipo double que refleja el total acumulado de ganancias para esa fecha. El constructor de la clase inicializa estos atributos, asegurando que cada instancia de Ganancia tenga una fecha y un monto total de ganancias.

La clase proporciona métodos getter para acceder a los atributos fecha y totalGanancia, permitiendo obtener la información registrada. El método añadirGanancia permite sumar un nuevo monto de ganancia al total existente, con una verificación para evitar valores negativos que podrían indicar errores en el cálculo. Además, el método toString ofrece una representación textual de la instancia de Ganancia, mostrando la fecha y el total de ganancias en un formato legible.

```java
public class Ganancia {
    private LocalDate fecha;
    private double totalGanancia;

    // Constructor
    public Ganancia(LocalDate fecha, double totalGanancia) {
        this.fecha = fecha;
        this.totalGanancia = totalGanancia;
    }

    // Getters
    public LocalDate getFecha() {
        return fecha;
    }

    public double getTotalGanancia() {
        return totalGanancia;
    }

    // Método para añadir una ganancia al total existente
    public void añadirGanancia(double ganancia) {
        if (ganancia < 0) {
            throw new IllegalArgumentException("La ganancia no puede ser negativa.");
        }
        this.totalGanancia += ganancia;
    }

    // Método para mostrar la información de la ganancia
    @Override
    public String toString() {
        return "Fecha: " + fecha + ", Total Ganancia: $" + totalGanancia;
    }
}
```

Para agregar productos en la aplicación, se tiene que introducir el nombre del producto, el precio, el stock, una imagen relacionada y su categoría, como se observa en la siguiente imagen.

```java
private void agregarProducto(ActionEvent event) {
    try {
        String nombre = this.txtNombre.getText();
        double precio = Double.parseDouble(this.txtPrecio.getText());
        int stock = Integer.parseInt(this.txtStock.getText());

        if (rutaImagen == null || rutaImagen.isEmpty()) {
        Alert alert = new Alert(Alert.AlertType.WARNING);
        alert.setHeaderText(null);
        alert.setTitle("Advertencia");
        alert.setContentText("Por favor, selecciona una imagen.");
        alert.showAndWait();
        return;
        }

        if (categoriaSeleccionada == null || categoriaSeleccionada.isEmpty()) {
            Alert alert = new Alert(Alert.AlertType.WARNING);
            alert.setHeaderText(null);
            alert.setTitle("Advertencia");
            alert.setContentText("Por favor, selecciona una categoría.");
            alert.showAndWait();
            return;
        }

        ProductoDeVenta producto;
        switch (categoriaSeleccionada) {
            case "Bebida" -> producto = new Bebida(nombre, precio, stock, rutaImagen);
            case "Comida rápida" -> producto = new ComidaRápida(nombre, precio, stock, rutaImagen);
            case "Postre" -> producto = new Postre(nombre, precio, stock, rutaImagen);
            case "Snack" -> producto = new Snack(nombre, precio, stock, rutaImagen); // Crea esta clase si es necesario
            case "Fruta" -> producto = new Fruta(nombre, precio, stock, rutaImagen);
            case "Otro" -> producto = new Otro(nombre, precio, stock, rutaImagen);
            default -> throw new IllegalArgumentException("Categoría desconocida: " + categoriaSeleccionada);
```

Una vez que se ingresan todos estos campos, se verifica que el producto no exista en el inventario de la aplicación, si no es así entonces se añade el producto en la aplicación y en la tabla de productos.

```
            if (!this.listaProductos.contains(producto)){

                this.appBarPoliEats.agregarProductoDerecha(producto);

                this.listaProductos.add(producto);
                this.tblCategorias.setItems(listaProductos);

            } else {
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setHeaderText(null);
                alert.setTitle("Error");
                alert.setContentText("El producto ya existe");
                alert.showAndWait();
            }

        } catch (NumberFormatException e) {
            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.setHeaderText(null);
            alert.setTitle("Error");
            alert.setContentText("Formato incorrecto");
            alert.showAndWait();
        }
    }
```

Para modificar un producto, se debe seleccionar uno de los que se muestran dentro de la tabla, una vez escogido el producto a modificar se puede cambiar cualquier atributo o característica que tenga el producto, menos la categoría, como se observa en la siguiente imagen.

```
@FXML
private void modificarProducto(ActionEvent event) {
    ProductoDeVenta producto = this.tblCategorias.getSelectionModel().getSelectedItem();

    if (producto == null){
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setHeaderText(null);
        alert.setTitle("Error");
        alert.setContentText("Debe seleccionar un producto");
        alert.showAndWait();
    }
    else {
        try {
            String nombre = this.txtNombre.getText();
            double precio = Double.parseDouble(this.txtPrecio.getText());
            int stock = Integer.parseInt(this.txtStock.getText());

            if (rutaImagen == null || rutaImagen.isEmpty()) {
                Alert alert = new Alert(Alert.AlertType.WARNING);
                alert.setHeaderText(null);
                alert.setTitle("Advertencia");
                alert.setContentText("Por favor, selecciona una imagen.");
                alert.showAndWait();
                return;
            }

            if (categoriaSeleccionada == null || categoriaSeleccionada.isEmpty()) {
                Alert alert = new Alert(Alert.AlertType.WARNING);
                alert.setHeaderText(null);
                alert.setTitle("Advertencia");
                alert.setContentText("Por favor, selecciona una categoría.");
                alert.showAndWait();
                return;
```

Estas modificaciones se guardan en una variable auxiliar, y en caso de que el producto modificado no coincida con ningún producto que exista ya dentro de la aplicación,

entonces se crea el producto, de la misma manera que con el método de creación, como se indica en la siguiente imagen.

```
ProductoDeVenta auxProducto;
switch (categoriaSeleccionada) {
    case "Bebida" -> auxProducto = new Bebida(nombre, precio, stock, rutaImagen);
    case "Comida rápida" -> auxProducto = new ComidaRápida(nombre, precio, stock, rutaImagen);
    case "Postre" -> auxProducto = new Postre(nombre, precio, stock, rutaImagen);
    case "Snack" -> auxProducto = new Snack(nombre, precio, stock, rutaImagen); // Crea esta clase si es necesario
    case "Fruta" -> auxProducto = new Fruta(nombre, precio, stock, rutaImagen);
    case "Otro" -> auxProducto = new Otro(nombre, precio, stock, rutaImagen);
    default -> throw new IllegalArgumentException("Categoría desconocida: " + categoriaSeleccionada);
}

if (!this.listaProductos.contains(auxProducto)){

    this.appBarPoliEats.modificarProductoDelInventario(producto, auxProducto);

    producto.setNombre(auxProducto.getNombre());
    producto.setPrecio(auxProducto.getPrecio());
    producto.setStock(auxProducto.getStock());
    producto.setRutaImagen(auxProducto.getRutaImagen());

    this.tblCategorias.refresh();
} else {
    Alert alert = new Alert(Alert.AlertType.ERROR);
    alert.setHeaderText(null);
    alert.setTitle("Error");
    alert.setContentText("El producto ya existe");
    alert.showAndWait();
}

catch (NumberFormatException e) {
```

Para eliminar un producto, se debe seleccionar uno de la tabla que se muestra en la intefaz, una vez seleccionado mediante un botón se elimina el producto tanto de la tabla como de la aplicación, como se visualiza en la siguiente imagen.

```
@FXML
private void eliminarProducto(ActionEvent event) {
    ProductoDeVenta producto = this.tblCategorias.getSelectionModel().getSelectedItem();

    if (producto == null){
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setHeaderText(null);
        alert.setTitle("Error");
        alert.setContentText("Debe seleccionar un producto");
        alert.showAndWait();
    }
    else {
        this.appBarPoliEats.eliminarProductoDelInventario(producto);

        this.listaProductos.remove(producto);
        this.tblCategorias.refresh();
    }
}
```

Cada vez que se salga de esta ventana con el botón de regresar, se actualizan los archivos txt que contienen los productos, de modo que al cargar de nuevo el programa se evidencie todos los cambios realizados, como se aprecia en la siguiente imagen.

```
private void regresarInterfasListaPedidos(ActionEvent event) {
    actualizarDatosProductos();
    MetodosFrecuentes.cambiarVentana((Stage) btnRegresar.getScene().getWindow(), "/Presentation/PantallaDelAdministrador.fxml", "Inicio");
}


private void actualizarDatosProductos() {
    // Lista de archivos de productos y sus tipos correspondientes
    String[][] archivosProductos = {
        {"src/Data/ProductosDeVenta/ProductosBebidas.txt", "Bebida"},
        {"src/Data/ProductosDeVenta/ProductosComidaRapida.txt", "ComidaRápida"},
        {"src/Data/ProductosDeVenta/ProductosPostre.txt", "Postre"},
        {"src/Data/ProductosDeVenta/ProductosSnacks.txt", "Snack"},
        {"src/Data/ProductosDeVenta/ProductosFruta.txt", "Fruta"},
        {"src/Data/ProductosDeVenta/ProductosOtro.txt", "Otro"}
        // Agrega más archivos y tipos según sea necesario
    };

    for (String[] archivoYTipo : archivosProductos) {
        actualizarArchivoConProductos(archivoYTipo[0], archivoYTipo[1]);
    }
}

private void actualizarArchivoConProductos(String archivo, String tipoProducto) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(archivo))) {
        for (ProductoDeVenta producto : listaProductos) {
            if (producto.getClass().getSimpleName().equals(tipoProducto)) {
                writer.write(producto.getNombre());
                writer.newLine();
                writer.write(String.valueOf(producto.getPrecio()));
                writer.newLine();
                writer.write(String.valueOf(producto.getStock()));
                writer.newLine();
                writer.write(producto.getRutaImagen()); // Escribe la dirección de la imagen
                writer.newLine();
            }
        }
```

Además, una parte del código se encarga de filtrar los productos según la categoría que se desea visualizar con más detalle, de modo que el administrador tenga la posibilidad de navegar de una manera más amigable en esta ventana, como se indica en la siguiente imagen.

```
private void seleccionarCategoria(String categoria) {
    this.categoriaSeleccionada = categoria;
    btnMenuCategorias.setText("Categoría: " + categoria);
}

private void filtrarPorCategoria(String categoria) {
    btnFiltroCategorias.setText("Filtro: " + categoria);
    listaFiltrada.setPredicate(producto -> producto.getCategoria().getCategoriaString().equals(categoria));
    tblCategorias.setItems(listaFiltrada);
    tblCategorias.refresh();
}

private void mostrarTodos() {
    btnFiltroCategorias.setText("Mostrar Todos");
    tblCategorias.setItems(listaProductos);
    tblCategorias.refresh();
}
```

Para poder presentar los productos disponibles y que los mismos sean seleccionados para crear un pedido se presentan las diferentes Listas y HashMap que permitirán ir viendo este proceso de manera dinámica mediante este va cambiando.

```
@Override
public void initialize(URL url, ResourceBundle rb) {
    //Inicializar Observable
    productImageMap = FXCollections.observableHashMap();
    ingresarSetCellValueFactory(this.colNombre, "nombre");
    ingresarSetCellValueFactory(this.colCantidad, "stock");
    ingresarSetCellValueFactory(this.colValor, "precio");

    productosSeleccionados = FXCollections.observableArrayList();
    ingresarSetCellValueFactory(this.colProducto, "nombre");
    ingresarSetCellValueFactory(this.colPrecio, "precio");
    this.colImagen.setCellValueFactory(cellData -> {
        ProductoDeVenta product = cellData.getValue();

        Image image = productImageMap.get(product);
        ImageView imageView = new ImageView(image);
        imageView.setFitHeight(50);
        imageView.setFitWidth(88);
        //imageView.setPreserveRatio(true);
        return new SimpleObjectProperty<>(imageView);
    });

    pedidoGenerado = FXCollections.observableArrayList();
    ingresarSetCellValueFactory(this.colValorFinalAPagar, "valorTotalAPagar");
    ingresarSetCellValueFactory(this.colNombreFinal, "nombreDelCliente");

    //Cargar Categorías
    txtCategoriaBebida.setOnAction(e -> seleccionarCategoria("Bebida", Categoría.BEBIDA));
    txtCategoriaComidaRápida.setOnAction(e -> seleccionarCategoria("Comida rápida", Categoría.COMIDA_RÁPIDA));
    txtCategoriaPostre.setOnAction(e -> seleccionarCategoria("Postre", Categoría.POSTRE));
    txtCategoriaSnack.setOnAction(e -> seleccionarCategoria("Snack", Categoría.SNACK));
    txtCategoriaFrutas.setOnAction(e -> seleccionarCategoria("Snack", Categoría.FRUTA));
    txtCategoriaOtros.setOnAction(e ->seleccionarCategoria("Comida rápida", Categoría.OTRO));
```

Una vez que yo he seleccionado una categoría la misma se carga con todos los productos que contienen y se muestran en una paginación para su posterior selección.

```
private void mostrarProductosPorPágina(Categoría tipoDeCategoría) {

    paginaDeProductos.setPageFactory(new Callback<Integer, Node>() {
@Override
public Node call(Integer pageIndex) {
    stockReal = productosCargados.getDato(paginaDeProductos.getCurrentPageIndex()).getStock();
    nombreReal = productosCargados.getDato(paginaDeProductos.getCurrentPageIndex()).getNombre();
    precioReal = productosCargados.getDato(paginaDeProductos.getCurrentPageIndex()).getPrecio();
    String rutaReal = productosCargados.getDato(paginaDeProductos.getCurrentPageIndex()).getRutaImagen();
    switch (tipoDeCategoría) {
            case Categoría.BEBIDA:
                productoMostradoAux = new Bebida(nombreReal, 0, 0, rutaReal);
                productoAuxGuardar = new Bebida(nombreReal, 0, 0, rutaReal);
                break;
            case Categoría.COMIDA_RÁPIDA:
                productoMostradoAux = new ComidaRápida(nombreReal, 0, 0, rutaReal);
                productoAuxGuardar = new ComidaRápida(nombreReal, 0, 0, rutaReal);
            break;
            case Categoría.POSTRE:
                productoMostradoAux = new Postre(nombreReal, 0, 0, rutaReal);
                productoAuxGuardar = new Postre(nombreReal, 0, 0, rutaReal);
                break;
            case Categoría.SNACK:
                productoMostradoAux = new Snack(nombreReal, 0, 0, rutaReal);
                productoAuxGuardar = new Snack(nombreReal, 0, 0, rutaReal);
                break;
            case Categoría.OTRO:
                productoMostradoAux = new Otro(nombreReal, 0, 0, rutaReal);
                productoAuxGuardar = new Otro(nombreReal, 0, 0, rutaReal);
                break;
            case Categoría.FRUTA:
                productoMostradoAux = new Otro(nombreReal, 0, 0, rutaReal);
                productoAuxGuardar = new Otro(nombreReal, 0, 0, rutaReal);
                break;
            default:
                throw new IllegalArgumentException("Categoría desconocida: " + categoriaSeleccionada);
        }
```

En nuestro proyecto, la gestión de las ganancias es un aspecto crucial que nos permite medir el éxito financiero y la viabilidad económica de nuestras operaciones. A través de la clase Ganancia, controlamos y registramos de manera precisa las entradas monetarias asociadas a fechas específicas. Esto no solo nos ayuda a llevar un registro organizado de los ingresos diarios, sino que también nos permite realizar análisis detallados para identificar patrones de ventas, evaluar el rendimiento en diferentes periodos y tomar decisiones informadas. Al implementar métodos que garantizan la integridad de los datos, como el control para evitar valores negativos en las ganancias, aseguramos que la información almacenada sea confiable y útil para la toma de decisiones estratégicas. Además, la capacidad de añadir nuevas ganancias al total existente facilita un seguimiento continuo y actualizado de los ingresos, lo que es fundamental para la planificación financiera y el éxito sostenido del proyecto.

```java
package BusinessLogic;
import java.time.LocalDate;

public class Ganancia {
    private LocalDate fecha;
    private double totalGanancia;

    // Constructor
    public Ganancia(LocalDate fecha, double totalGanancia) {
        this.fecha = fecha;
        this.totalGanancia = totalGanancia;
    }

    // Getters
    public LocalDate getFecha() {
        return fecha;
    }

    public double getTotalGanancia() {
        return totalGanancia;
    }

    // Método para añadir una ganancia al total existente
    public void añadirGanancia(double ganancia) {
        if (ganancia < 0) {
            throw new IllegalArgumentException("La ganancia no puede ser negativa.");
        }
        this.totalGanancia += ganancia;
    }

    // Método para mostrar la información de la ganancia
    @Override
    public String toString() {
        return "Fecha: " + fecha + ", Total Ganancia: $" + totalGanancia;
    }
}
```

E. RESULTADOS EJECUCION

Al principio se presentó una pantalla de carga, como se muestra en la siguiente imagen.

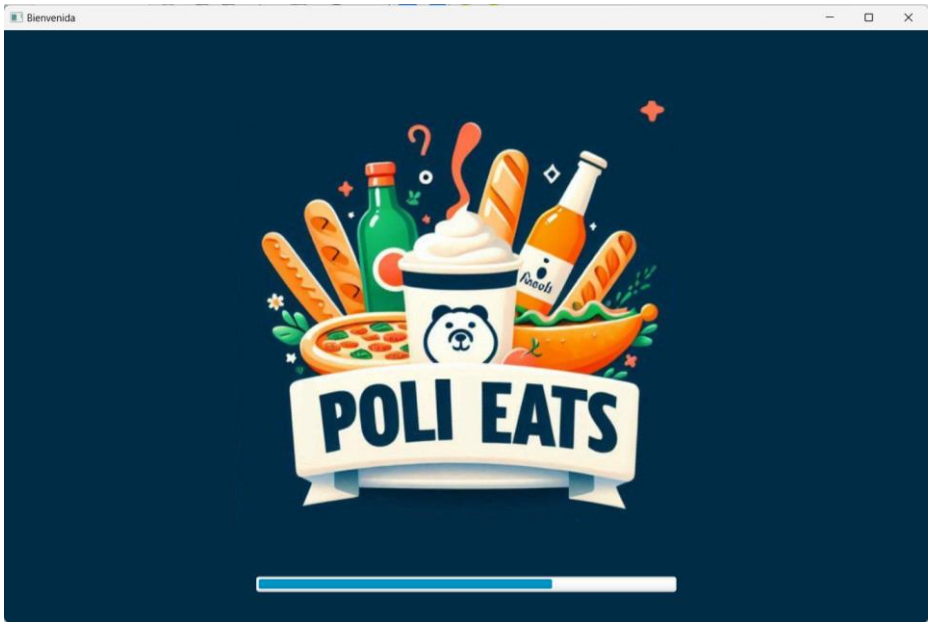


Ilustración 1 Pantalla de carga de la aplicación

En el inicio del programa, el administrador tiene que loguearse con su usuario y contraseña, en caso de no tener una cuenta, este puede crear una, como se indica en la siguiente imagen.

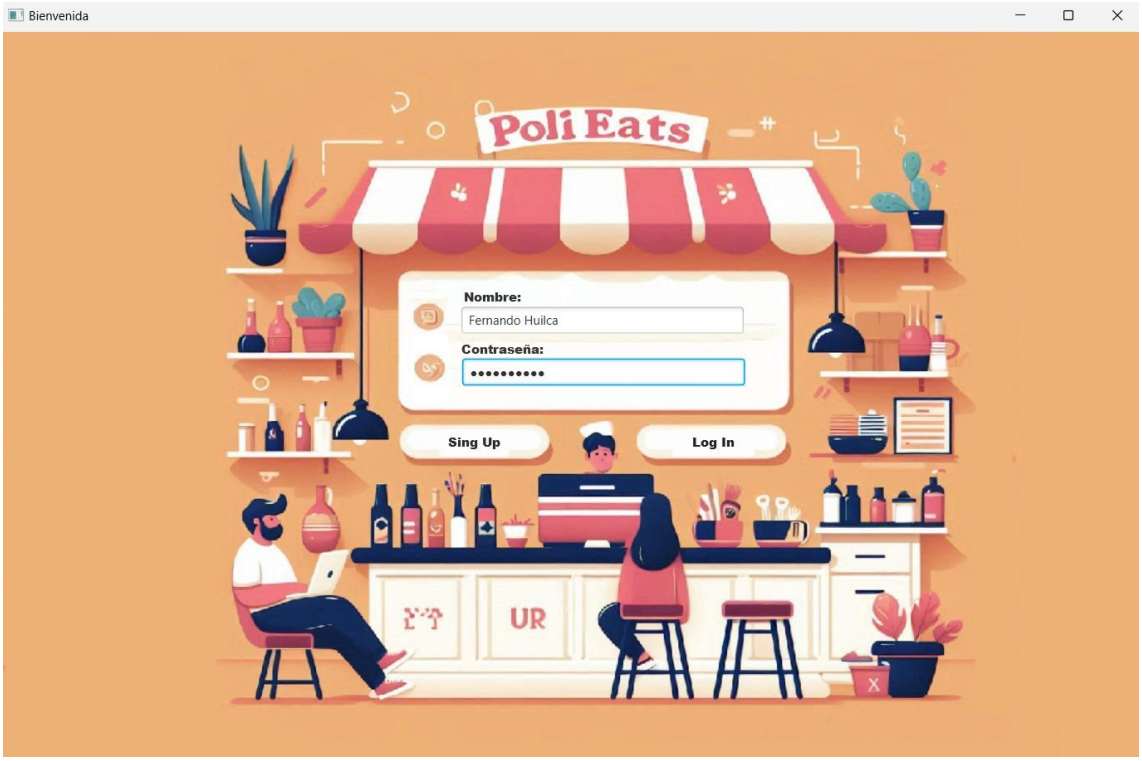


Ilustración 2 Login para administradores

A continuación, se muestra la pantalla de solicitud para agregar un nuevo usuario de administrador, aquí tiene que colocar su nombre, su correo electrónico y su contraseña, como se observa en la siguiente imagen.

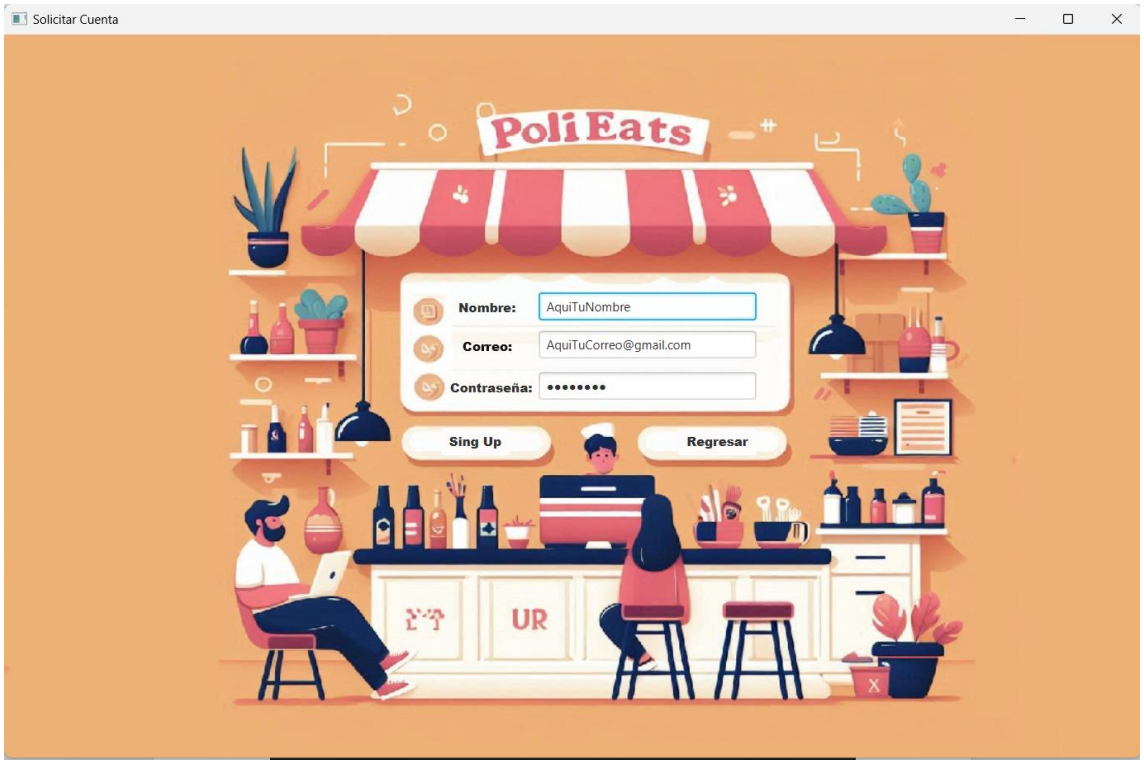
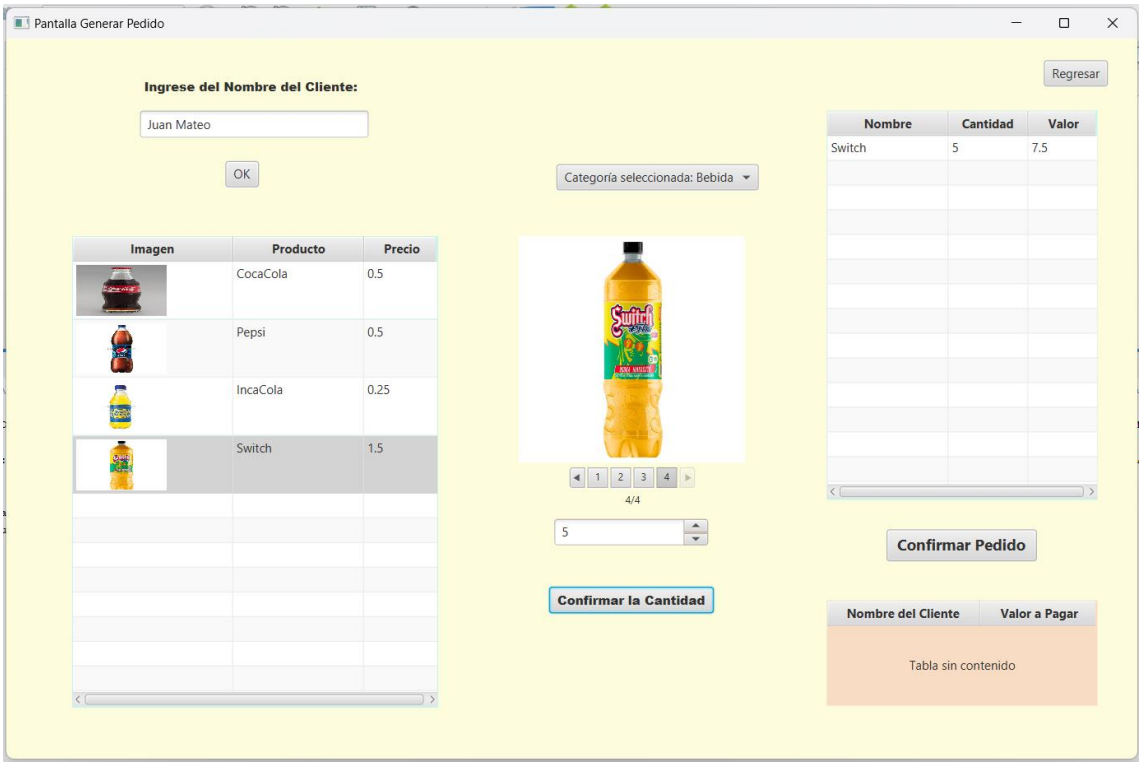


Ilustración 3 Sign up para administradores

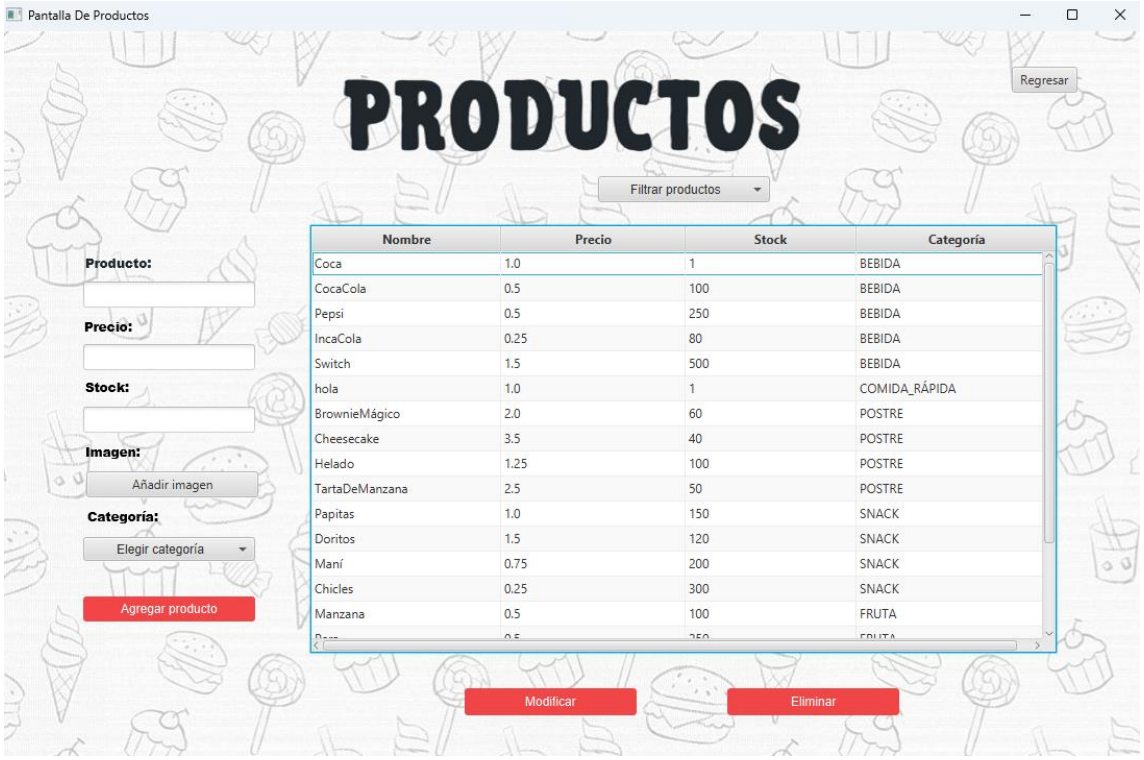
Una vez que se agrega un nuevo usuario, se tiene que confirmar un código de verificación, esto para validar que el correo sea real. El administrador tiene 3 intentos para validar su correo electrónico. En caso de no cumplir con este requerimiento, la cuenta no se crea.

Cuando el administrador logra ingresar al programa, tiene algunas opciones que puede realizar, como añadir un pedido, eliminarlo en el momento en que se atiende ese pedido, que como está funcionando la lógica de un FIFO, el primer pedido que se genera en el programa debe ser el primer pedido que debe ser eliminado puesto que ya fue atendido. Además puede dirigirse al apartado de productos, que se refiere al inventario que tiene el sitio. Por último, también puede observar sus ganancias. Todas estas opciones mencionadas, se explicarán más adelante.

En la siguiente pantalla se muestra la funcionalidad de generar un pedido, a la izquierda están todas las opciones disponibles, en la pantalla del medio se muestra el producto seleccionado teniendo la capacidad de configurar la cantidad de este y una vez que estoy de acuerdo con la cantidad lo confirmo y se coloca en la pantalla de la derecha y cuando he seleccionado todos los productos que necesito por pedido entonces va a la parte inferior generándose así el pedido.



En la siguiente imagen, se presenta el contenido de la pantalla de productos, aquí se puede visualizar de mejor manera el funcionamiento de esta. En esta parte del programa, el administrador tiene la posibilidad de añadir un producto, realizar modificaciones de cualquier producto e inclusive eliminar uno que ya no disponga o no genere ganancias. Además, se le presenta una tabla con el contenido principal de cada producto, y de ser el caso, puede filtrar los productos con la finalidad de encontrar de manera más eficaz y eficiente el producto que sea de su agrado obtener cierta información, como se muestra en la siguiente imagen.



Además, el administrador tiene la posibilidad de visualizar las ganancias generadas por día. Esto le permitirá determinar si su negocio se dirige hacia un buen camino o si tiene que decidir entre seguir con el negocio o cerrarlo, como se observa en la siguiente imagen.

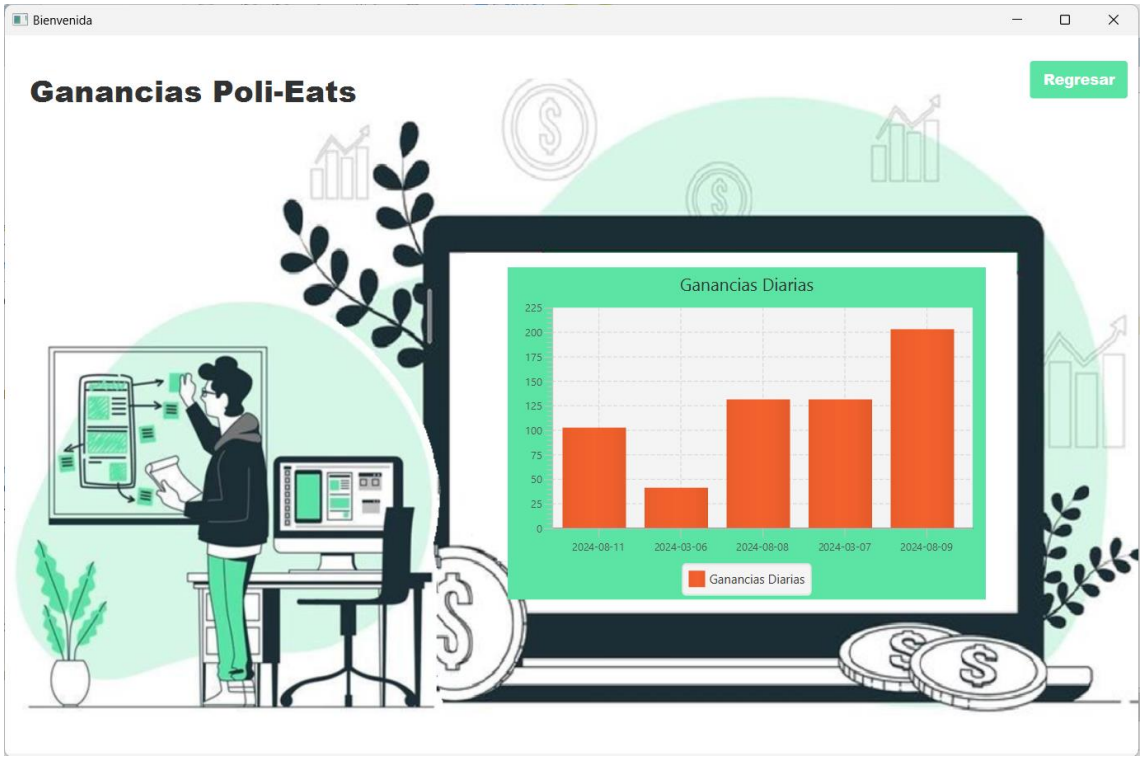


Ilustración 6 Pantalla de ganancias diarias en el aplicativo

F. BIBLIOGRAFÍA:

[1] Refactoring Guru, "Patrón Singleton", [Online]. Available: https://refactoring.guru/es/design-patterns/singleton. [Accessed: 13-Aug-2024].

[2] J. Sánchez, "MVC: Model-View-Controller explicado," *CódigoFacilito*, 15-Jun-2022. [En línea]. Disponible en: https://codigofacilito.com/articulos/mvc-model-view-controller-explicado. [Accedido: 13-Aug-2024].

G. Anexos

Ilustración 4 Reunión 2, definición de clases y niveles de abstracción

Ilustración 5 Reunión división de responsabilidades

Ilustración 6 Reunión avances en el proyecto

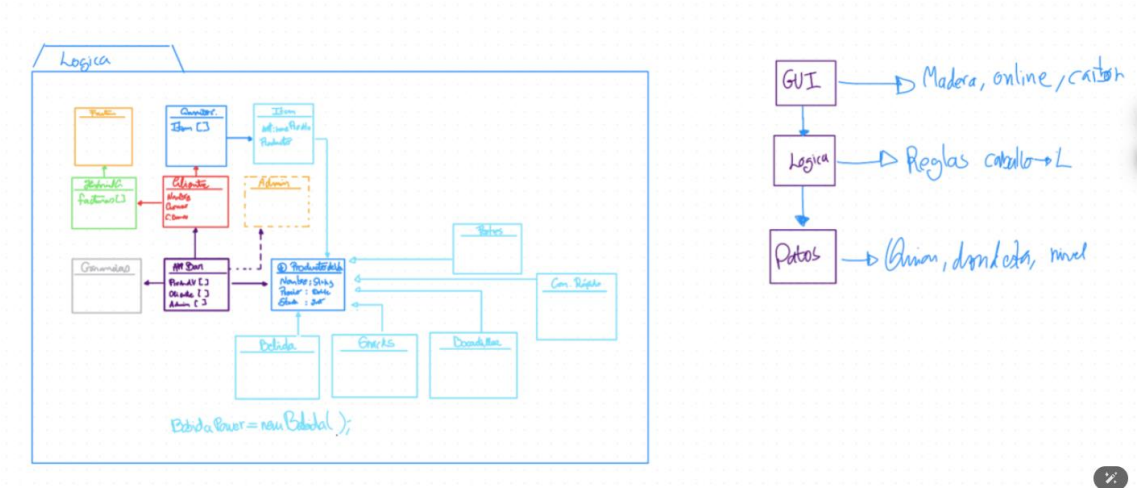


Ilustración 7 División de clases, versión inicial

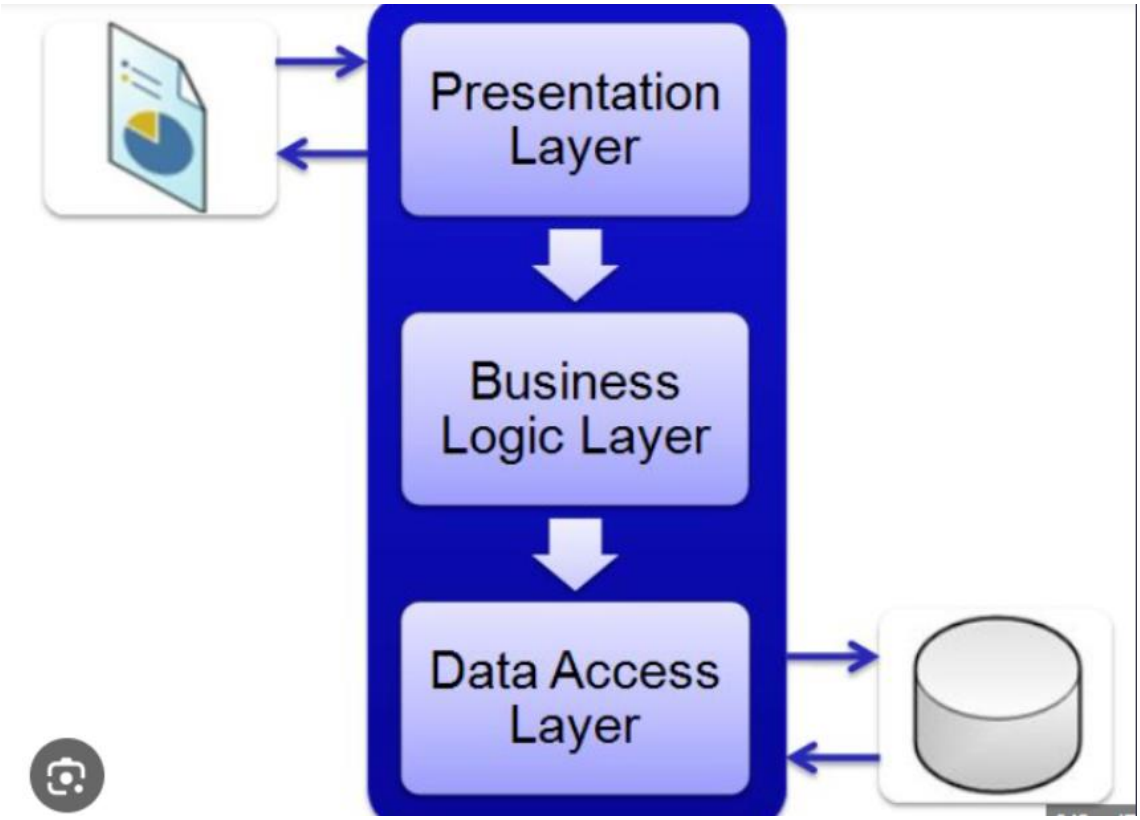


Ilustración 8 Conceptos para la organización conceptual del proyecto