PROGRAMACIÓN ORIENTADA A OBJETOS CON ACCESO A BBDD

DISEÑO UML

MIEMBROS

Katiane Coutinho Rosa
Nadia Igual Bravo
Xavier Melo Pardo
Daniel Boj

Grupo - Cirice

Producto 1. Diseño UML

Enlace GitHub

Para realizar el seguimiento de las tareas que vamos realizando, tanto a nivel individual como grupal, compartimos el enlace a nuestro repositorio de GitHub:
https://github.com/DanielBoj/UOC-POO-con-acceso-a-BBDD

Análisis de los requisitos según el caso práctico

A través del análisis del caso de uso, Podemos identificar los requisitos de nuestros Stakeholders:

- **Problema general:** Crear una aplicación de escritorio que maneje las ventas a través de comercio electrónico de la empresa Online Store. (La aplicación sirve como alternativa en el backend de la aplicación móvil).
    - A la aplicación de escritorio accederá un usuario que será el empleado encargado de gestionar las ventas de la empresa.
    - La aplicación debe manejar tres **entidades** → Artículos, clientes y pedidos.
    - La aplicación tendrá un interfaz visual a través del cual interactuará el usuario.
    - El interfaz se basará en un **menú principal** desde el cual se podrá acceder a los submenús para interactuar con cada entidad → Vista de artículos, vista de clientes y vista de pedidos.
    - La aplicación debe permitir realizar operaciones CRUD sobre las entidades y aplicará ciertas restricciones siguiendo la lógica de negocio.
- **Funcionalidades requeridas:** Las funciones principales requeridas por los stakeholders se pueden entender como una aplicación de algunas de las funciones CRUD.
    - **Gestión de artículos:**
        - Crear artículo
        - Listar todos los artículos.
    - **Gestión de clientes**
        - Crear cliente
        - Listar todos los clientes
        - Listar clientes de tipo Estándar
        - Listar clientes de tipo Premium
    - **Gestión de pedidos**
        - Crear un pedido
        - Eliminar un pedido
        - Listar pedidos pendientes
        - Listar pedidos enviados
    - **Restricciones**
        - Eliminar pedido → Solo podrán eliminarse pedidos que no hayan sido enviados.
        - Opción de filtrado por cliente para el listado de pedidos.

        - Opción de filtrado por estado del pedido para el listado de pedidos.
        - Crear pedido → Si el cliente no existe, permitirá crear un nuevo cliente.
        - Calculo del precio de envío → Se creará un método para calcular los gastos de envío segun el PVP, unidades pedidas, tipo de cliente y coste de envío del articulo.
- **Funcionalidades auxiliares**
    - Generación de códigos únicos para clientes Premium y para los productos.
    - Funciones de filtrado de tipos de cliente.
    - Funciones de filtrado de estado de pedido.
    - Funciones de búsqueda para las tres entidades.
    - Automatización de la asignación del estado de los productos → Incluye calcular si el pedido ha sido enviado teniendo en cuenta el tiempo de prepareción de este.

Identificación de los actores

Fijándonos en el análisis de los requisitos de la aplicación, podemos identificar un único actor que será del tipo **usuario** y representará al empleado encargado de la gestión de las ventas de la empresa en el backend.

Identificación de los casos de uso

Atendiendo a las funcionalidades identificadas en el análisis de requisitos, podemos identificar los casos de uso de nuestra aplicación. Se han englobado de forma general según el tipo de entidad, así quedan más esquematizados y podrían realizarse subesquemas de casos de uso de ser necesario.

- Gestionar artículos
    - Añadir artículo
    - Mostrar artículos
- Gestionar clientes
    - Añadir clientes
    - Mostrar clientes
        - Filtrar clientes
- Gestionar pedidos
    - Realizar pedido → Incluye añadir cliente
        - Enviar pedido* (cambia el estado del pedido) → Extiende realizar pedido
        - Eliminar pedido → Extiende realizar pedido
    - Mostrar pedidos pendientes
        - Filtrar pedido → Realizará la actualización del estado de los pedidos.
    - Listar pedidos
        - Filtrar pedidos

*La funcionalidad de enviar pedido, en un escenario real, podría implicar la implementación de un nuevo módulo para la app que gestionará los envíos. En nuestro caso, simplemente se tratará de, como se ha indicado en el análisis de las funcionalidades auxiliares, **cambiar el estado del pedido**.

Antes de pasar a la identificación de las clases, realizamos una pequeña introducción del patrón de diseño MVC ya que servirá para detector qué clases dependientes de este patrón implementará nuestra aplicación.

Patrón Modelo-Vista-Controlador

El patrón de diseño de Modelo-Vista-Controlador permite implementar en Java, y los demás leguajes del paradigma de Programación Orientada a Objetos, un modelo de tareas CRUD manteniendo la independencia y autonomía de la lógica de negocio y la interfaz visual. Con ello, nos aseguraremos mantener la independencia del trabajo, reduciendo así el alcance de cualquier incidencia y mejorando los costes de implementación de mejoras y nuevas funcionalidades y el mantenimiento de las funcionalidades ya creadas.
El patrón MVC, como su propio nombre indica, se basa en la separación de la estructura del programa en tres subestructuras.

Modelo

Define las estructuras de datos, que normalmente representan a las tablas, en caso de trabajar con una BBDD de tipo relacional, o los documentos, si trabajamos con mongo, u otras estructuras de datos de nuestra BBDD.
Los cambios en las estructuras del modelo sirven para actualizar las vistas con la información que se presentan al usuario. Para gestionar todas las operaciones sobre este módulo principal, usaremos una **clase a modo de controlador interno**. Esta clase manejará todas las operaciones sobre las entidades, su estructuración en listas y la conexión a la base de datos.
En nuestro caso, tenemos 3 entidades principales; **artículos, clientes y pedidos**. Como estas podrán instanciarse con cardinalidad múltiple, usaremos colecciones para gestionarlas, en este caso, ArrayList de Java. Con el fin de generalizar estas listas, usaremos una **clase génerica Listas** para establecer el modelo de métodos y atributos del ArrayList. Por otro lado, tal y como se ha visto en los requisitos, la clase cliente se especializará en dos subclases, **estándar y premium** a través de la herencia.
Por ultimo, para gestionar la conexión a la base de datos de forma aislada, usaremos la clase **conexión** para implementar toda la lógica interna del proceso. Además, para generalizar los tipos de datos y métodos usaremos las siguientes **clases e interfaces**:

- **Dirección (clase auxiliar)** → Implementa el modelo de datos del atributo dirección de la clase Cliente.
- **HashCode (interfaz)** → Implementa el método generalizado para crear códigos únicos para los códigos de artículo y de cliente premium.

Por último, comentar que creemos que puede resultar útil usar la interfaz **comparable** de Java para implementar búsquedas de objetos.

Vista

Este módulo principal estructura el diseño y la presentación de la interfaz de usuario mediante la cual se interactuará con la aplicación. La hemos esquematizado en base a un **menu principal** que será el encargado de crear y gestionar los submenús: **vista de artículos, vista de clientes** y **vista de pedidos**. De este modo, cada clase de Vista representa una ventana de la GUI.

Controlador

Por último, la estructura del controlador se encarga de enrutar y comunicar las otras dos estructuras, ya que tanto vista como modelo deben permanecer ciegas una a la otra. Así, contiene la lógica de administración del Sistema conectando Modelo y Vista.
Para ello, se implementa un módulo **controlador**, que sirve como controlador principal de la aplicación y que podemos entender como una API que se encarga de comunicar con las peticiones del usuario, manipular los datos de modelo y actualiza las vistas con las respuestas obtenidas. Además, dentro de este módulo principal encontramos la clase **main** que se encarga de iniciar y dirigir el flujo de ejecución de la aplicación.

Identificación de clases

Atendiendo tanto a las entidades identificadas en el análisis de requisitos como a las clases necesarias para estructurar el diseño MVC, hemos identificado las siguientes clases.

- **Modelo**
  - **Datos:** Actúa como controlador interno y gestiona todas las operaciones CRUD sobre las entidades y la conexión a la base de datos.
  - **Listas (clase genérica):** Generaliza todos los atributos y métodos de las colecciones que usaremos para almacenar y gestionar las entidades.
  - **Conexión:** Independiza la lógica para realizar la conexión a la BBDD.
  - **Artículo:** Implementa la lógica para manejar la entedidad artículo.
  - **Cliente:** Implementa la lógica para manejar la entidad cliente.
    - **ClientePremium:** Subtipo de cliente.
    - **ClienteEstandar:** Subtipo de cliente.
  - **Pedido:** Implementa la lógica para manejar la entidad pedido.
  - **Dirección:** Prototipa el atributo dirección de la clase Cliente.
  - **HashCode(interfaz):** Prototipa los métodos para obtener un código único.
- **Vista**
  - **MenuPrincipal:** Genera la venta del menú principal en el GUI y sirve para manejar, a modo de controlador interno, al resto de vistas.
  - **VistaArticulo:** Crea y maneja la ventana de gestión de artículos.
  - **VistaCliente:** Crea y maneja la ventana de gestión de clientes.
  - **VistaPedido:** Crea y maneja la ventana de gestión de pedidos.
- **Controlador**
  - **Main:** Inicia y maneja el flujo de ejecución del programa y crea las instancias necesarias.

- **Controlador:** Controlador principal del programa, gestiona la lógica de relaciones entre los módulos Modelo y Vista y maneja la lógica de administración de la aplicación.

Sobre los métodos

Teniendo en cuenta que pueden surgir nuevos requisitos o especificaciones en los siguientes productos, entendemos que los métodos que hemos identificado en el diagrama de clases **tienen una naturaleza flexible y que podemos añadir nuevos métodos a las clases según las necesidades y requisitos espeíficos de cada producto.** De igual forma, hay métodos que pueden faltar ya que aun no sabemos como implementar funcionalidades como la conexión a la BBD o la creación de vistas.

Modelo de Casos de Uso

Dado las dimensiones del modelo, se incluye en un archivo externo:
producto1(FP058)_NombreCirice_DiagramaCasosDeUso.pdf

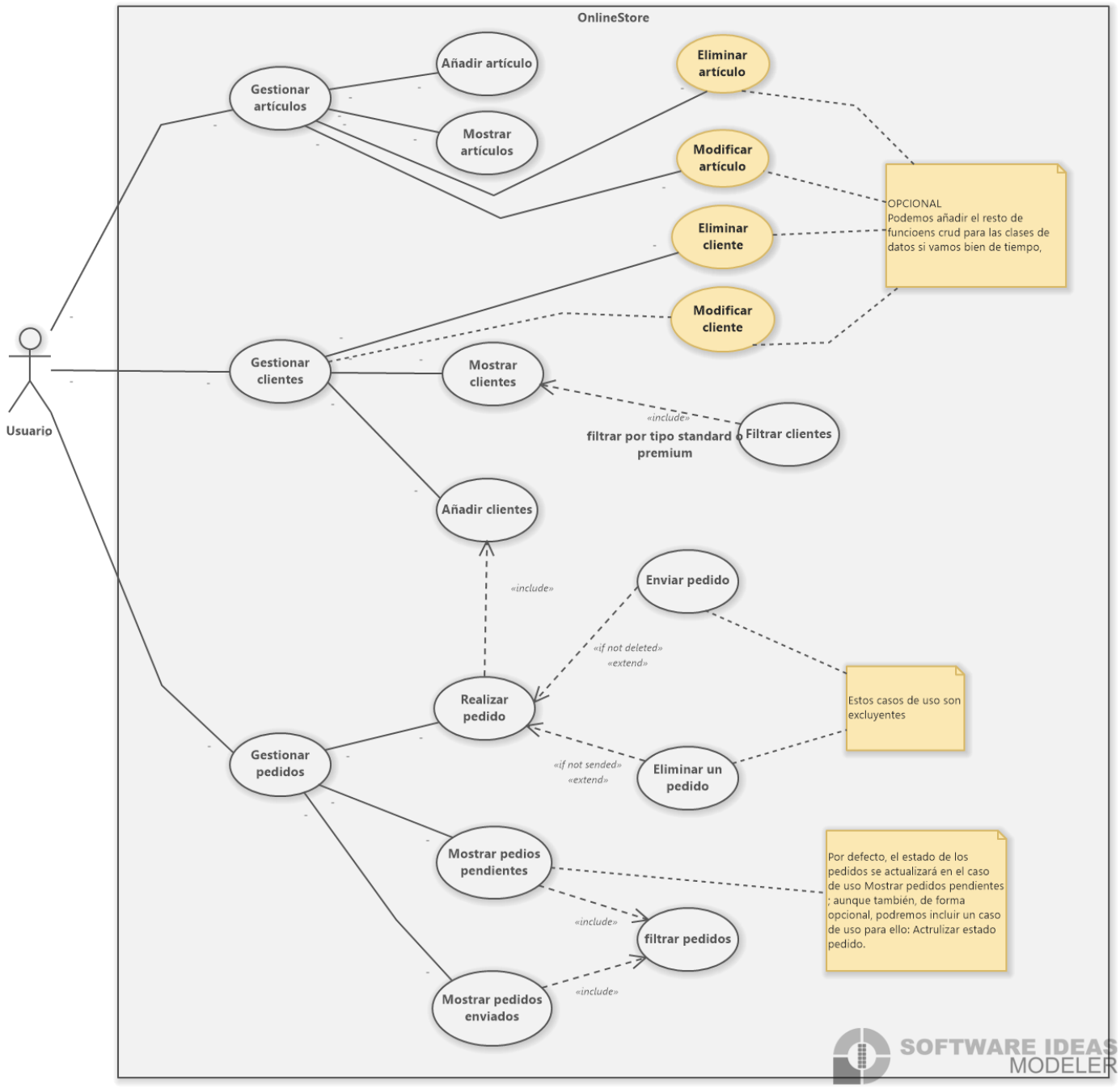
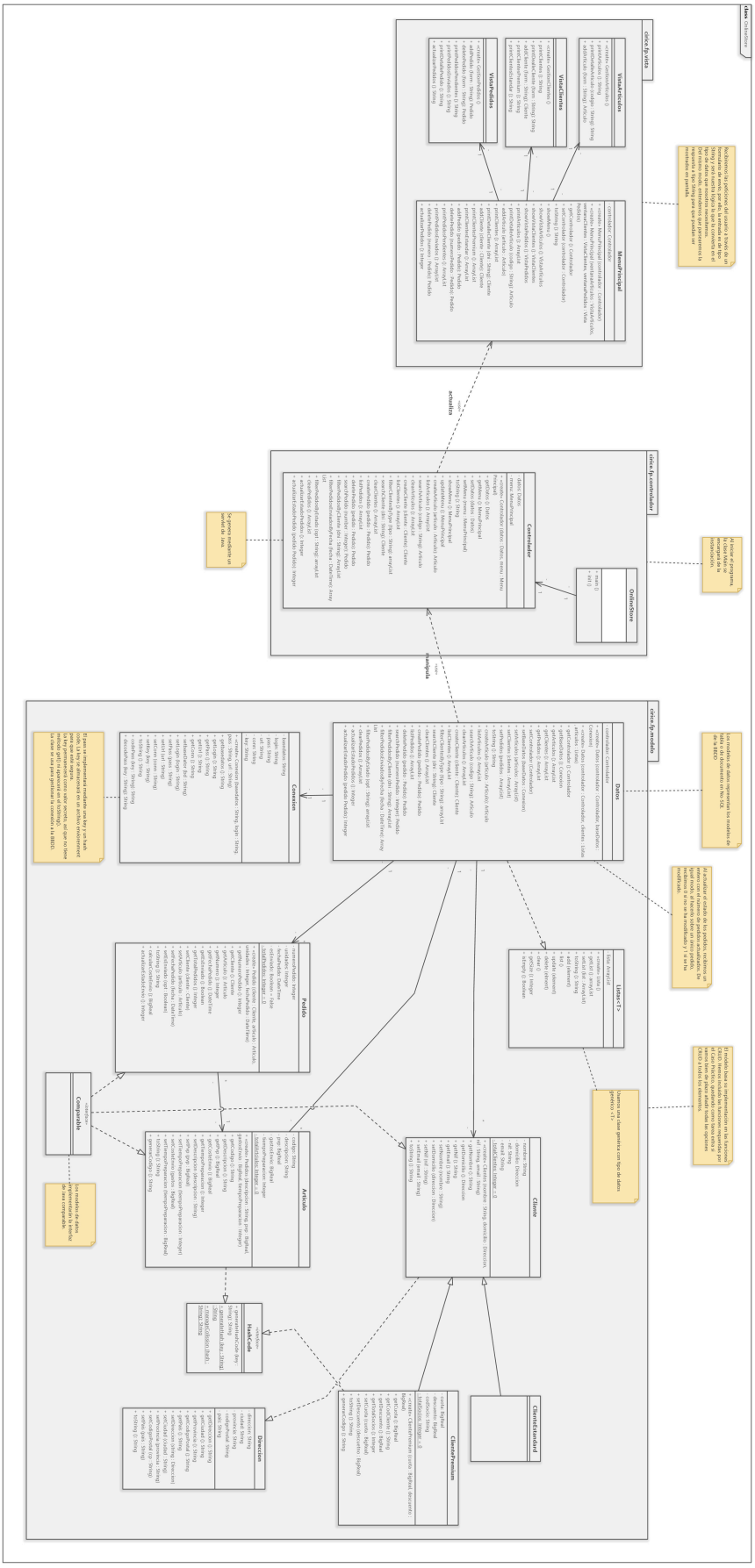


Diagrama de clases

Dado las dimensiones del diagrama, se incluye en un archivo externo:

producto1(FP058)_NombreCirice_DiagramaClases_BojCobos_Daniel.pdf



Bibliografía

Henao., C. (s. f.). *Ejemplo Modelo Vista Controlador*. http://codejavu.blogspot.com/2013/06/ejemplo-modelo-vista-controlador.html

MVC - Glosario de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN. (2022, 5 diciembre). https://developer.mozilla.org/es/docs/Glossary/MVC

pildorasinformaticas. (2017a, abril 4). *Curso Java. MVC I. Vídeo 248*. YouTube. https://www.youtube.com/watch?v=H5A5eXbyPxM

pildorasinformaticas. (2017, 11 abril). *Curso Java. MVC III. Vídeo 250*. YouTube. https://www.youtube.com/watch?v=z1PREOrTKRE

Charly Cimino. (2022, 16 mayo). *ABSTRACT en JAVA MÉTODOS abstractos*. YouTube. https://www.youtube.com/watch?v=6iF5Gs-nKVs

B. (2021, 11 noviembre). *Comparator and Comparable in Java*. Baeldung. https://www.baeldung.com/java-comparator-comparable

MVC Architecture in Java - Javatpoint. (s. f.). www.javatpoint.com. https://www.javatpoint.com/mvc-architecture-in-java

pildorasinformaticas. (2017a, abril 4). *Curso Java. MVC I. Vídeo 248*. YouTube. https://www.youtube.com/watch?v=H5A5eXbyPxM