ECART-DELIVERY COMPANY

PRACTICA # 2 (PYTHON)

ASIGNATURA

Programación Orientada a Objetos

INTEGRANTES

Sebastian Cadavid, Angel Pimienta, y Santiago Giraldo

PROFESOR

Jaime Alberto Guzman Luna

Universidad Nacional de Colombia

Sede Medellín

2023

Contenido

Capítulo 1

Sección 1.1: Descripción general

🛍️🛒 ECart
"Empower Your Passion, Share Your Creations"

📃 Descripción

ECart (Carrito Electrónico) es una aplicación de E-commerce para la compra y venta de productos creados por los usuarios. ECart permite convertir hobbies como Crochet, Origami, Dibujo, etc., en fuentes de ingresos rentable, facilitando la comercialización de dichos productos misceláneos ofreciendo hosting, exposición, manejo de trámites y Delivery rentable.

Sección 1.2: Análisis

Análisis: los usuarios pueden crear "tiendas" (Store) virtuales de manera individual o conjunta, en las cuales pueden vender sus creaciones. Además, los usuarios pueden ser recomendados y/o explorar por su cuenta otras tiendas. Nuestro modelo de negocios consiste en facilitar el transporte las mercancías. Nuestros Delivery's llegan a los hogares de los vendedores y entregan los paquetes a los destinatarios correspondientes.

Sección 1.3: Diseño

Diseño: por un lado, cada Usuario puede tener o hacer parte de varias tiendas (Store) virtuales, en las cuales se exponen los Product(os). Cada Product a la venta puede tener Coupon(es) y Reviews, que pueden ser usados por usuarios que vayan a comprar los productos. Como cada User puede comprar y vender, todos tienen un ShoppingCart, un historial de compras (Purchases), órdenes de compra (Order) y pagos opcionalmente fraccionables por cada orden (Payment). Por otro lado, los Admin de la aplicación, pueden crear repartidores (Delivery) y administrar los recursos monetarios de la compañía.

Sección 1.4: Implementación

Implementación: la clase abstracta Person es la clase padre de la que heredan User, Delivery y Admin y tienen sus propios métodos dinámicamente ligados. Cada uno tiene métodos para hacer sus funcionalidades (comprar, vender, etc.). Todo se realiza mediante la TUI (Textual User Interface) que tiene varios métodos de con sobrecarga (Utils) y constantes en forma de un Enum para los banners (Banners). La mayoría de los atributos de cada una de estas clases están aislados a sus paquetes o a la misma clase, dependiendo del contexto de cada una.

Capítulo 2

Sección 2.1: Diagrama de clases

Capítulo 3

Sección 3.1: Clases y métodos abstractos

public abstract Class Entity:

La clase Entity es una clase abstracta que sirve como base para representar entidades en un sistema. Cada entidad tiene un nombre y una dirección asociada. Además, se implementa una lógica para gestionar direcciones únicas en un rango de 0 a 100.

Tiene atributo de addresses: Una lista de direcciones que se comparte entre todas las instancias de la clase Entity como también tiene la función de validar si la dirección ingresada por el usuario es correcta teniendo en cuenta las condiciones expuestas en el manual de usuario.

Cuenta específicamente con los siguientes métodos:
__init__(self, name: str, address: Tuple[int, int]) -> None: Constructor de la clase. Se encarga de inicializar una entidad con un nombre y una dirección. Verifica la disponibilidad de la dirección y lanza una excepción si ya está en uso.

is_address_available(self, _address) -> bool: Método que verifica si una dirección está disponible para ser asignada a una entidad. Comprueba si la dirección está dentro del rango permitido (0 a 100) y verifica que no este en uso por otra entidad.

Cuenta con los métodos getters y setters para cada atributo

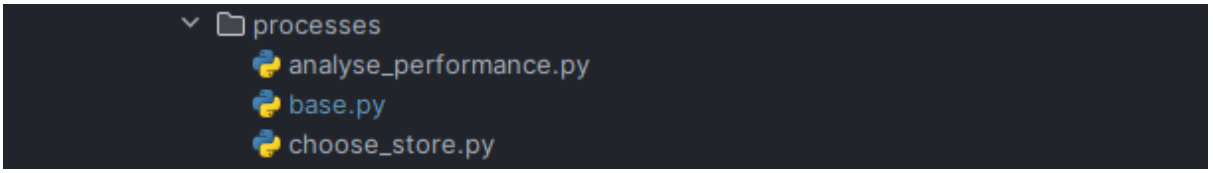
get_name(self) -> str: Método que devuelve el nombre de la entidad.

set_name(self, name: str) -> None: Método que actualiza el nombre de la entidad.

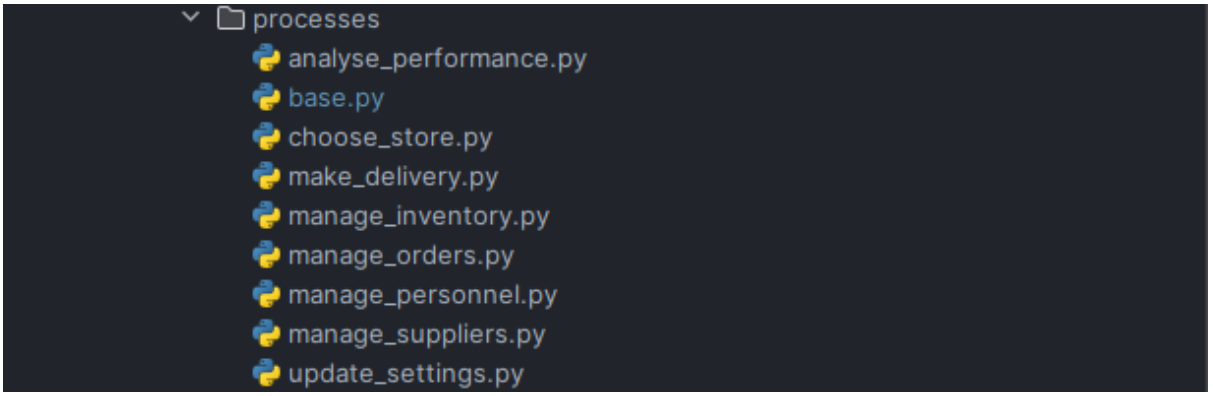
get_address(self) -> Tuple[int, int]: Método que devuelve la dirección de la entidad.

set_address(self, address) -> None: Método que actualiza la dirección de la entidad.

Abstract class Base



La clase Base es una clase abstracta que actúa como una base para la creación de la interfaz de cada una de las funcionalidades expuestas en el menú de “Procesos y Consultas” . Al heredar de tk.Frame, proporciona una estructura básica para la interfaz gráfica, y al heredar de ABC, declara métodos abstractos que deben ser implementados por las clases hijas que este caso son las clases en el directorio de processes:



Cuenta con los siguientes atributos:

master: El widget principal al que está asociado este marco.

title: El título de la interfaz gráfica.

description: Una descripción asociada a la interfaz gráfica.

Métodos Abstractos

```python
@abstractmethod
def __init__(self, master: tk.Misc, title: str, description: str) -> None:
    super().__init__(master)
```

__init__(self, master: tk.Misc, title: str, description: str) -> None: Constructor de la clase. Inicializa el marco con el widget principal, título y descripción.

```python
@abstractmethod
def setup_ui(self):
```

setup_ui(self): Método abstracto que debe ser implementado por las clases hijas para configurar la interfaz gráfica. Este método se encarga de definir la disposición de los widgets y cualquier configuración adicional necesaria para la interfaz como los frames, labels, o cualquier otro requerimiento que se necesite para las diferentes funcionalidades.

Sección 3.3: Herencia

Package Entites

Clase Base (Entity): La clase Admin, Delivery en el paquete de Entites. La clase Entity proporciona propiedades y métodos comunes a todas las entidades que

permiten las correcta gestión e interacción entre las diferentes clases, adicionalmente la clase store también extiende de Entity

Subclase (Admin, Delivery, Store): Cada una de las clases hijas también puede agregar sus propias propiedades y métodos específicos para representar características que son exclusivas de cada una de las subclases.

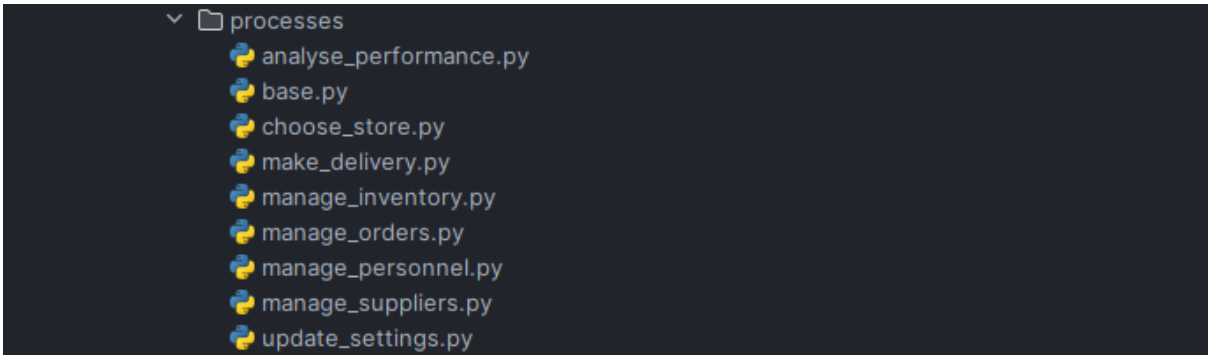
Package Helpers

Clase field_frama (tk.Frame): Esta hereda o implimenta Frame de tkinter para extender las funcionalidades este y personalizar su comportamiento especifico y así representar una variedad de productos finales como formularios y diferentes widgets.

Clase scrollable_text (tk.Text): Esta clase adquiere todas las propiedades y métodos de Text en tkinter, la razón para esto es que sirve para asegurar entre sus métodos la correcta inicialización y desplazamiento vertical del texto.

Lo cual también es genérico para las clases Inicio y principal la cuales heredan de tk.Frame.

Package Processes

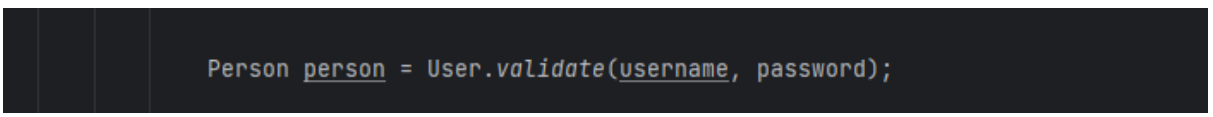


Clase Base (Base): En esta clase se encuentra la plantilla general de la cual podrán heredar todas las demás clases que requieran tener entre sus métodos la esquematización de sus propios espacios o frames teniendo en cuenta las necesidades y propósitos para las cuales son hechas.
Analyse_perfomance, choose_store, make_delivery, manage_inventory, manage_personnel, manage_suppliers y update_settings hacen parte las clases hijas de base.py

Clase Errors

Clase ErrorAplicacion(Exception): Es una clase derivada de la clase base que incorporada o implementa Exception.
Tiene un constructor que toma dos parámetros (kind y msg) para proporcionar información sobre el tipo y el mensaje del error.
Se utiliza como base para otras clases que representan tipos específicos de errores.
Esta superclase proporciona propiedades y métodos fundamentales lo cuales son cruciales tanto para la administración como para la autenticación de las entidades y por otra parte ofrece la flexibilidad necesaria para que cada tipo de entidad, a través de sus subclases, adapte y personalice sus propias particularidades.

Sección 3.4: Ligadura Dinámica



Curabitur nec euismod leo, id vulputate nunc. Quisque posuere rhoncus mauris in venenatis. Phasellus in auctor dui. Maecenas rhoncus felis ligula, nec feugiat diam molestie sit amet. In neque odio, eleifend eu leo vitae, elementum porta sem. Morbi ornare eros vel lacus lobortis, quis vestibulum ex efficitur. Praesent et viverra metus. Cras nec sem eget quam fermentum gravida a vitae metus. Fusce vel faucibus lectus, vitae feugiat risus. Proin vitae massa aliquet, malesuada massa posuere, vestibulum velit.

Sección 3.5: Atributos y métodos de clase

Clase Delivery

```
1 usage  secadavida
@classmethod
def find(cls, name: str, arr):
```

Este método de clase se utiliza para buscar las diferentes instancias de Delivery por nombre en un lista de entregas (arr).

```
@classmethod
def create(cls, name: str, address: Tuple[int, int], workplace: Store,
           arr: list):
```

Este método se utiliza para crear una nueva instancia de Delivery en caso de no haber otra con el mismo nombre en la lista de entregas (arr) además notemos el uso de cls para referirnos a la clase.

Clase Product

```
new *
@classmethod
def find(cls, name: str, arr):
```

Aunque los métodos de clase de la clase producto se parezcan a los de Delivery ambos métodos sirven para cosas completamente diferentes, la cuales vamos a recalcar, en este método de find se usa para buscar instancias de Product por el nombre dentro de una lista de productos

```
2 usages (2 dynamic)  new *
@classmethod
def create(cls, name: str, price: float, quantity: int, description: str):
```

Tiene el mismo nombre del método de Delivery, sin embargo si nos fijamos mejor en los parámetros que recibe podemos notar la clara diferencia que hay entre ambos métodos, por otra parte tiene como función crear instancias de producto con los parámetro que le son proporcionados.

Clase Store

```
1 usage   secadavida +2
class Store(Entity):
    instances = []
```

Este es un método de clase de store que tiene como funcionalidad almacenar las distintas instancias u objetos que se crean asociados a tiendas en la aplicación.

```
    @classmethod
    def get(cls) -> list:
```

La función de este método de clase es devolver una lista de instancias ya creadas de la clase en cuestión

```
    @classmethod
    def find(cls, name: str):
```

El método find se usa para buscar instancias como lo indica su nombre de la cual pueden surgir dos acciones a tener en cuenta, la primera es devolver la primera instancia que haya cumplido con la condición, la segunda es devolver none en caso de no existir

```
    @classmethod
    def create(cls, name: str, address: Tuple[int, int], tag: Tags,
               description: str):
```

Este método de clase puede crear y/o devolver una nueva instancia de la clase si no hay otra instancia con el mismo nombre en caso contrario devolverá none si ya existe.

```
    @classmethod
    def get_instances(cls):
```

Este método de clase se usa para devolver todas las instancias creadas de la clase store.

Clase MsgboxWrapper

```
@classmethod
def show(cls, message_type, message, parent=None) -> Any:
```

Esta clase se usar como contenedor para simplificar el uso de las diferentes ventanas de dialogo de tkinter la cual tiene un método de clase que mostrara un cuadro de mensaje dependiendo del tipo de mensaje proporcionado:

message_type: Tipo de mensaje, puede ser "i" para información, "w" para advertencia, "e" para error, "aq" para pregunta, o "ay" para confirmación.

message: El mensaje que se mostrará en el cuadro de mensaje.

parent: El widget padre para el cuadro de mensaje.

Sección 3.6: Constantes

Clase Tags

Considerando que en Python no hay constantes o al menos no con la misma consistencia que en java estas hacen la representación de constantes las cuales están referidas a la clase Tags, específicamente a los iconos que representan las diferentes categorías o etiquetas, de manera que están representadas por emojis, acá se puede ver algunos ejemplos:

```python
class Tags(Enum):
    ALIMENTOS = "🍅"
    HOGAR = "🏠"
    LIMPIEZA = "🧹"
    ELECTRODOMESTICOS = "📺"
    JUGUETES = "🐎"
```

Clase Commons

```
__init__.py
commons.py
utils.py
```

En esta clase se definen varias constantes que representan las distintas fuentes tipográficas, la cuales contienen valores que representan el nombre de la fuente, el tamaño y opciones adicionales referida a la negrilla y cursiva, lo cual nos ayuda a mantener la coherencia y reutilización del código en la aplicación.

```python
TEXT_FONT = ("Broadway", 12)
TEXT_FONT_B = ("Broadway", 12, "bold")
TEXT_FONT_I = ("Broadway", 12, "italic")
TEXT_FONT_BI = ("Broadway", 12, "bold italic")
HEADER_FONT = ("Courier", 17, "bold")
DESC_FONT = ("Courier", 14, "bold")
```

Por otra parte, también cuenta con un diccionario relacionado a los autores de la aplicación, cada autor esta representado por una clave y una tupla con el nombre, edad y su correspondiente descripción, en este apartado se muestran de manera genérica:

```python
AUTHORS = {
    "angel": ("Angel", "3", "Desarrollador de software apasionado"),
    "sebastian": ("Sebastian", "3", "Desarrollador de software apasionado"),
    "rodrigo": ("Rodrigo", "3", "Desarrollador de software apasionado"),
    "santiago": ("Santiago", "3", "Desarrollador de software apasionado")
}
```

Sección 3.7: Encapsulamiento

Package Merchadise

Acá citamos las clases de Product y Store las cuales contienen claros ejemplos sobre el encapsulamiento.

Clase Product

```
Angell Pimienta's *
class Product:
    new *
    def __init__(self, name: str, price: float, quantity: int, description: str):

        self._name = name
        self._price = price
        self._quantity = quantity
        self._historic_quantity = quantity
        self._description = description
```

Los atributos de clase _name, _price, _quantity, _historic_quantity, _description) se han marcado como privados (Por convención) mediante el uso del guion bajo al principio del nombre del atributo (_) lo cual nos sirve para tratarlos como privados y que su acceso desde afuera se debería evitar.

Clase Store

```python
1 usage  secadavida +2 *
class Store(Entity):
    instances = []

    secadavida +1 *
    def __init__(self, name: str, address: Tuple[int, int], tag: Tags,
                 description: str):
        super().__init__(name, address)
        self._description = description
        self._tag = tag
        self._balance = 0
        self._orders: list[Order] = []
        from ecart.gestorAplicacion.entites.delivery import Delivery
        self._deliveries: list[Delivery] = []
        self._products: list[Product] = []
```

De la misma manera que en Product los atributos privados están dados por la convención de guion bajo al iniciar el nombre del atributo lo cual nos indica a modo de buenas prácticas que no debería tener acceso ninguna clase fuera de él los siguientes atributos:
(_description, _tag, _balance, _orders, _deliveries, _products)

Package Transactions

En este paquete podemos observar la encapsulación en su clase order la cual cuenta atributos privados clave para mantener un código sencillo y bien organizado.

Clase Order

```python
2 usages  RBucheli8 *
class Order:
    instances = []

    new *
    def __init__(self, products: dict[str, int], destinationAddress,
                 origin_address):
        self._products = products
        self._destinationAddress = destinationAddress
        self._origin_address = origin_address
        self._totalPrice = 0
        self._delivered = False
        self._deliveryPrice = 0
        self._id = len(Order.instances)
```

En la clase Order podemos observar los principios del encapsulamiento en Python lo que nos permite traerla como ejemplo:
Los atributos de la clase (_products, _destinationAddress, _origin_address, _totalPrice, _delivered, _deliveryPrice, _id) los reconocemos como privados debido al uso de guiones bajos lo que nos permite tener cierto control adicional con el principio de encapsulamiento.

Otros ejemplos de encapsulamiento:

```
RBucheli8 *
class Delivery(Entity):
    new *
    def __init__(self, name: str, address: Tuple[int, int], workplace: Store):
        super().__init__(name, address)
        self._workspace = workplace
```

```
secadavida *
class Admin(Entity):
    secadavida
    def __init__(self, name, address):

        super().__init__(name, address)
        self._current_store: Store | None = None
```

Sección 3.8: Enumeración

Enum Tags

Enum: Tags es una enumeración, la cual cuenta con una colección de valores constantes llamados enumeradores.
Cada enumerador en este caso representa una etiqueta o categoría las cuales cuentan con nombres y valores asociados.
Cuenta con los siguientes métodos:

Método get_list(): Es un método de clase que devuelve una lista con todos los valores de las etiquetas en la enumeración lo cual facilita obtener una lista de todos los emojis asociados a las categorías.

Método get_entry_for_tag(): Este método toma un valor de la etiqueta como argumento y devuelve la constante de la enumeración que contiene ese valor y en caso de encontrarla devolverá none.

Ejemplo de uso: Las constantes de la enumeración se pueden acceder como atributos de la clase, por ejemplo, Tags.ALIMENTOS representa la categoría de alimentos y tiene el valor asociado "🍎".

Capítulo 4

Sección 4.1: Funcionalidad 1

Hacer una orden:
Inicio: La función placeOrder se llama desde una instancia de la clase User.

Obtener Carrito de Compras: Se obtiene el carrito de compras del usuario actual a través de la llamada this.getShoppingCart().

Validación del Pedido: Se inicia un proceso de validación del pedido, donde se recorren los productos y las cantidades en el carrito de compras.

Para cada producto en el carrito, se verifica si la cantidad ordenada es menor o igual a la cantidad disponible (esta verificación se realiza llamando a ShoppingCart.isProductAvailable(product, orderedQuantity)). Si un producto no tiene suficiente stock, se agrega a la lista troublesomeProducts con detalles sobre la falta de stock.
Verificación de Productos Problema: Se verifica si la lista troublesomeProducts contiene algún producto que no tiene suficiente stock. Si la lista no está vacía, se crea una instancia de Retval con un mensaje de error que incluye los productos problemáticos y se devuelve.

Cálculo del Precio Total: Si todos los productos en el carrito de compras tienen suficiente stock, se procede a calcular el precio total del pedido. Se recorren los productos en el carrito, y para cada producto se multiplica el precio unitario por la

cantidad ordenada. La suma de estos valores para todos los productos en el carrito da como resultado el precio total del pedido.

Actualización de Stock: Para cada producto en el carrito de compras, se actualiza la cantidad disponible restando la cantidad ordenada.

Creación de Pedido: Se crea una instancia de la clase Order que representa el pedido realizado. La instancia de Order se inicializa con los productos y cantidades del carrito, el usuario que realiza el pedido y el precio total.

Almacenamiento del Pedido: El pedido recién creado se agrega a la lista de pedidos del usuario (this.orders).

Vaciar Carrito de Compras: Finalmente, el carrito de compras se vacía llamando al método clearItems().

Retorno: Se devuelve una instancia de Retval que informa que el pedido se ha realizado con éxito y se proporciona un mensaje al usuario para indicar que debe realizar el pago después de la entrega.

Sección 4.2: Funcionalidad 2

Inicio: La función suggestProducts se llama desde una instancia de la clase User que representa al usuario que desea recibir recomendaciones de productos. La función toma como argumentos un arreglo de etiquetas (tags) que representan categorías de productos y un valor máximo de precio (maxPrice).

Inicializar Variables: Se inicializan variables, como recommendedProducts (una lista de productos recomendados), allStores (una lista de todas las tiendas disponibles en el sistema), userStores (las tiendas en las que el usuario es miembro) y availableStores (las tiendas que no son propiedad del usuario y, por lo tanto, están disponibles para recomendaciones).

Convertir Arreglo de Etiquetas a Lista: El arreglo tags se convierte en una lista de etiquetas llamada tagsList para facilitar la comprobación de etiquetas de productos.

Recorrer Todas las Tiendas: La función itera a través de todas las tiendas disponibles en el sistema (representadas por objetos Store) y verifica si el usuario es miembro de cada una.

Crear una Lista de Tiendas Disponibles: Las tiendas que no son propiedad del usuario se agregan a la lista availableStores.

Recorrer Tiendas Disponibles: La función itera a través de las tiendas disponibles en availableStores.

Recorrer Productos en Cada Tienda: Para cada tienda, la función itera a través de todos los productos en esa tienda (representados por objetos Product).

Verificar Etiquetas y Precio: Para cada producto, se verifica si sus etiquetas (product.getTag()) están contenidas en la lista tagsList y si el precio del producto es igual o inferior al valor maxPrice.

Agregar Producto Recomendado: Si el producto cumple con los criterios de etiquetas y precio, se agrega a la lista recommendedProducts.

Finalización: La función retorna la lista recommendedProducts, que contiene todos los productos que cumplen con los criterios de etiquetas y precio.

Sección 4.3: Funcionalidad 3
Enviar los productos por delivery:
Inicio: La función deliverOrder se llama desde una instancia de la clase User que representa a un repartidor.

Establecer Repartidor: La instancia del pedido (order) que se va a entregar se actualiza con el repartidor asignado. Esto se realiza llamando al método order.setDeliveryUser(this) para asignar al repartidor actual a la entrega del pedido.

Calcular Precio de Entrega: Se llama a la función getDeliveryPrice para calcular el precio de entrega del pedido. Esta función puede tener en cuenta diferentes factores, como la distancia o el método de entrega. El precio de entrega se almacena en la instancia del pedido (order) como deliveryPrice.

Actualizar Estado del Pedido: Se actualiza el estado del pedido para marcarlo como "entregado". Esto se hace llamando al método order.setDelivered(true).

Actualizar Precio Total del Pedido: El precio total del pedido se actualiza sumando el precio de entrega (deliveryPrice) al precio total original del pedido (order.getTotalPrice()). Esto se refleja en la instancia del pedido (order).

Retorno de Resultado: Se crea una instancia de Retval que informa que el pedido se ha entregado exitosamente. Se incluye un mensaje indicando el monto a pagar al repartidor (deliveryPrice) una vez que el usuario final realice el pago.

Sección 4.4: Funcionalidad 4

Inicio: La función getDeliveryPrice se llama desde una instancia de la clase User que representa el usuario final que recibirá el pedido. Además, se pasa un objeto Order que contiene información sobre los productos seleccionados en el pedido.

Inicializar Variables: Se inicializan variables, como selectedProducts (los productos seleccionados en el pedido), route (una lista de coordenadas de direcciones) y totalPrice (el precio total del servicio de entrega).

Obtener Dirección Actual: Se obtiene la dirección actual del usuario que llama a la función, representada por currentAddress, y se agrega a la lista de rutas route.

Recorrer Productos Seleccionados: La función itera a través de los productos seleccionados en el pedido y calcula el costo de entrega para cada producto en función de la distancia entre la dirección actual y la dirección del titular del producto.

Calcular el Precio del Producto: Para cada producto, se calcula el precio de entrega utilizando una tarifa estándar. La distancia entre la dirección actual y la dirección del titular del producto se divide por 5 y se multiplica por 2 para calcular el costo de entrega.

Actualizar la Ruta: Se agrega la dirección del titular del producto a la lista de rutas route. La dirección actual se actualiza para que coincida con la del titular del producto.

Calcular la Distancia y el Precio del Último Producto: Después de haber iterado a través de todos los productos, la función calcula la distancia y el precio de entrega desde el último producto hasta la dirección del usuario final. El proceso es similar al paso 5, utilizando la dirección del último producto y la dirección del usuario final.

Actualizar la Ruta con la Dirección del Usuario Final: La dirección del usuario final se agrega a la lista de rutas route.

Calcular el Precio Total: El precio total de la entrega se calcula sumando todos los precios calculados en los pasos anteriores.

Retorno del Resultado: La función retorna el precio total de la entrega.

Sección 4.5: Funcionalidad 5

Inicio: La función abonarOrder se llama desde una instancia de la clase User que representa un usuario que realiza un pago parcial en un pedido.

Calcular Monto a Abonar: La función verifica si el monto de abono (abono) es mayor que la cantidad pendiente por pagar en el pedido. Si es mayor, se ajusta el monto de abono para igualar la cantidad pendiente por pagar.

Verificar Saldo de la Cuenta Bancaria: La función verifica si el usuario tiene suficiente saldo en su cuenta bancaria para cubrir el monto de abono. Si el saldo es insuficiente, se devuelve un Retval con un mensaje de error.

Realizar Abono: Se llama al método orderToPay.abonar(toPay) para realizar el abono en el pedido. Esto disminuye la cantidad pendiente por pagar en el pedido.

Retirar Dinero de la Cuenta Bancaria: Se llama al método this.getBankAccount().withdraw(toPay) para deducir el monto del abono de la cuenta bancaria del usuario.

Verificar Pago Completo: Se verifica si el pedido ha sido completamente pagado. Si la cantidad pagada (orderToPay.getPayedSoFar()) es igual o mayor que el precio total del pedido (orderToPay.getTotalPrice()), se procede a realizar pagos adicionales a otras partes involucradas.

Realizar Pagos Adicionales: Si el pedido se ha pagado completamente, se realizan pagos adicionales a otras partes involucradas, como al repartidor (orderToPay.getDeliveryUser()) por la entrega y al titular de los productos seleccionados. Estos pagos se realizan llamando al método this.makePayment().

Retorno de Resultado: Se crea una instancia de Retval que informa que el abono se ha realizado con éxito. Si el pago ha cubierto completamente el pedido, también se informa sobre los pagos adicionales realizados a otras partes involucradas.

Capítulo 5

Sección 5.1: Diagrama - Estructura del Programa