3A. MODELADO Y PROGRAMACIÓN EN POO: HERENCIA, AGREGACIÓN Y COMPOSICIÓN EN PYTHON

Unidad 3: Programación Orientada a Objetos (POO)

Datos
Nombre: Keith Alexis Gutiérrez Ibarra
Docente: Adolfo Aldair Duque Borja
Fecha de elaboración: 28/11/2025

Introducción

En este ejercicio se aplicara la Programación Orientada a Objetos mediante el desarrollo de un sistema de ventas para una tienda de productos tecnológicos, donde a partir de una situación planteada se diseñaran las clases principales del sistema y se implementaran relaciones como herencia, agregación y composición, el objetivo de este ejercicio es construir una solución que sea organizada, reutilizable y fácil de mantener, aplicando los principios fundamentales como son abstracción, encapsulación y el uso de decoradores.

A lo largo del desarrollo se crearán usuarios con diferentes roles, productos, ventas y una tienda que sea capaz de gestionar todas las transacciones, además de representar el sistema mediante un diagrama UML que ayudara a visualizar la estructura y la forma en que cada clase interactúa con las otras, con este ejercicio se busca reforzar el uso de POO y mostrar cómo sus conceptos nos permiten resolver problemas reales de manera más clara y estructurada.

Desarrollo

Ejercicio

a) Análisis del problema: Para el ejercicio se nos piden las siguientes condiciones que debe tener el Código.

- Crear un diagrama UML acerca del funcionamiento del código.
- Modelar usuarios con herencia, creando una clase base abstracta (Usuario) de la cual deben derivar Cliente y Administrador, cada uno con atributos y comportamientos específicos.
- Implementar productos y ventas, donde cada Venta debe contener varios objetos de tipo Producto.
- Crear una clase Tienda encargada de administrar y almacenar todas las ventas realizadas
- Aplicar decoradores y métodos especiales, de manera que el sistema permita validar datos, mostrar información, mejorar el seguimiento y funcionalidad del código.
- Generar una ejecución final que muestre el funcionamiento completo del sistema, incluyendo el registro de usuarios, creación de productos, generación de ventas y almacenamiento de dichas ventas dentro de la tienda.

b) Diseño del algoritmo

Se organizan las clases para modelar el sistema de ventas y se definen las relaciones entre ellas según los principios de POO.

a) Se crea la clase Usuario como clase abstracta
b) Se importan las librerías como abc para definir métodos abstractos
c) Se establecen los atributos generales de la clase usuario, así como los métodos que deberán ser implementados por las clases hijas.

Se diseñan las clases Cliente y Administrador que heredan de Usuario.

a) Se inicializan con atributos propios de cada tipo de usuario.
b) Se sobrescriben los métodos abstractos heredados para definir su estructura específica.

Se diseña la clase Producto encargada de representar los artículos de la tienda.

a) Se definen atributos como nombre, precio y contador.
b) Se implementan métodos para validar información y mostrar detalles del producto.

Se crea la clase Venta mediante composición.

a) La clase recibe un objeto de la clase Cliente y una lista de objetos Producto.
b) Se agregan métodos para agregar productos, calcular el total y mostrar el contenido de la venta

Se diseña la clase Tienda que aplica agregación para gestionar ventas.

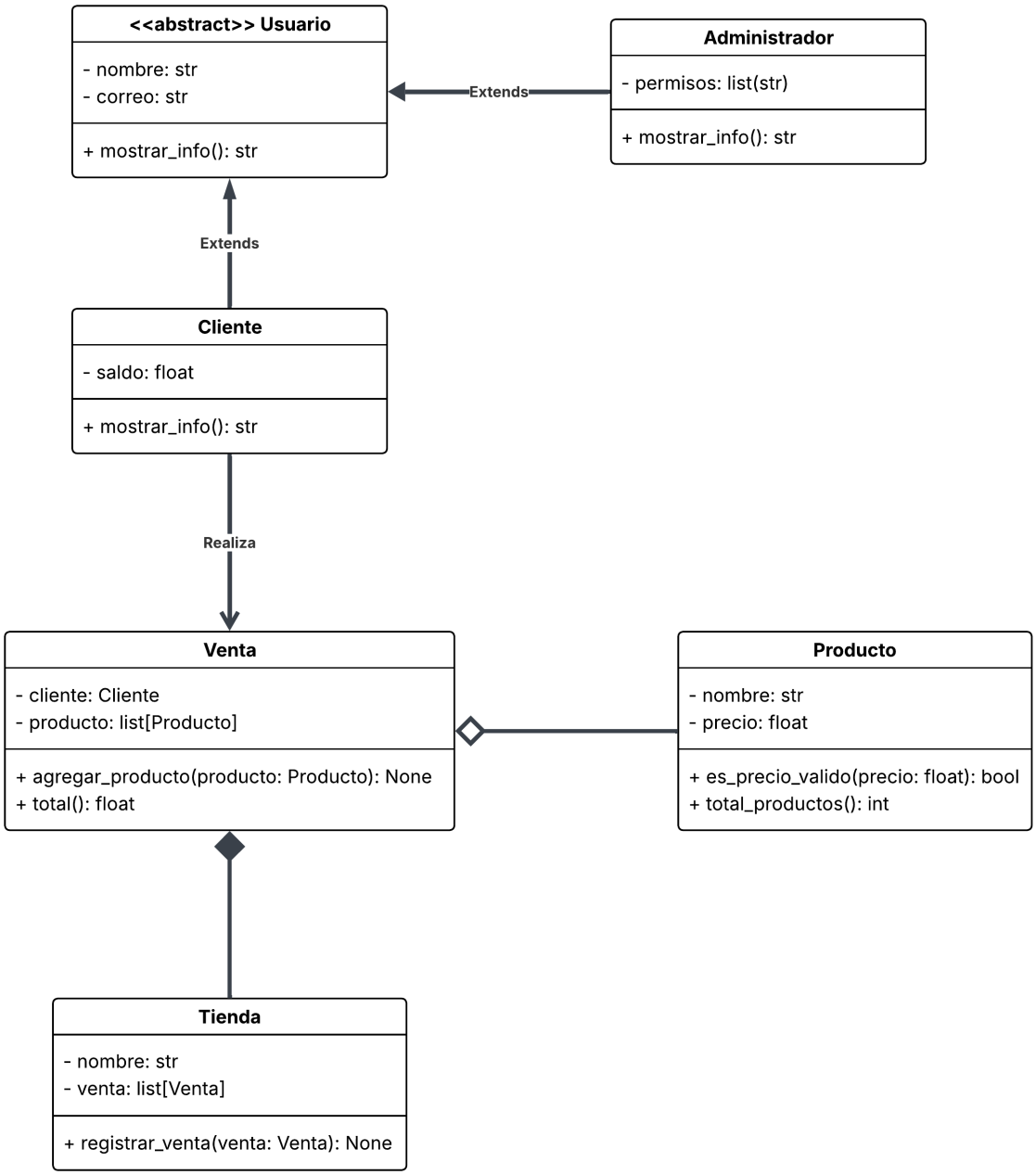
a) Se crea una lista interna donde se almacenan las ventas realizadas.
b) Se implementan métodos para registrar nuevas ventas.

Se desarrolla el archivo principal main.

a) Se instancian usuarios (Cliente).
b) Se crean varios productos con sus datos correspondientes.
c) Se generan ventas.
d) Se imprimen resultados mostrando la información requerida.

Se realiza la ejecución del archivo main instanciando los objetos necesarios.

Diagrama UML

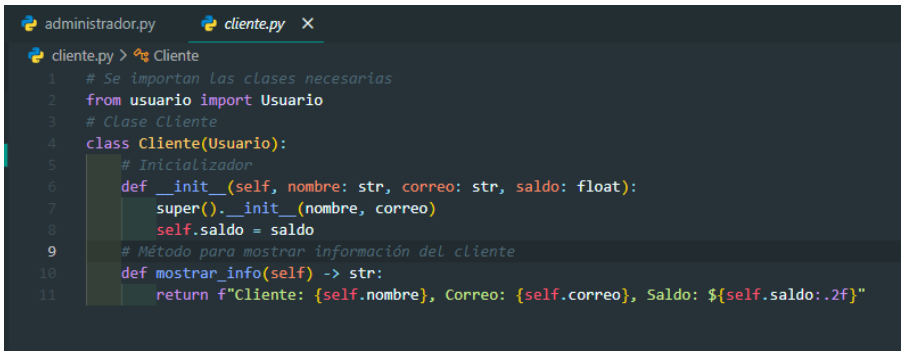


c) Código en Python

- Administrador.py

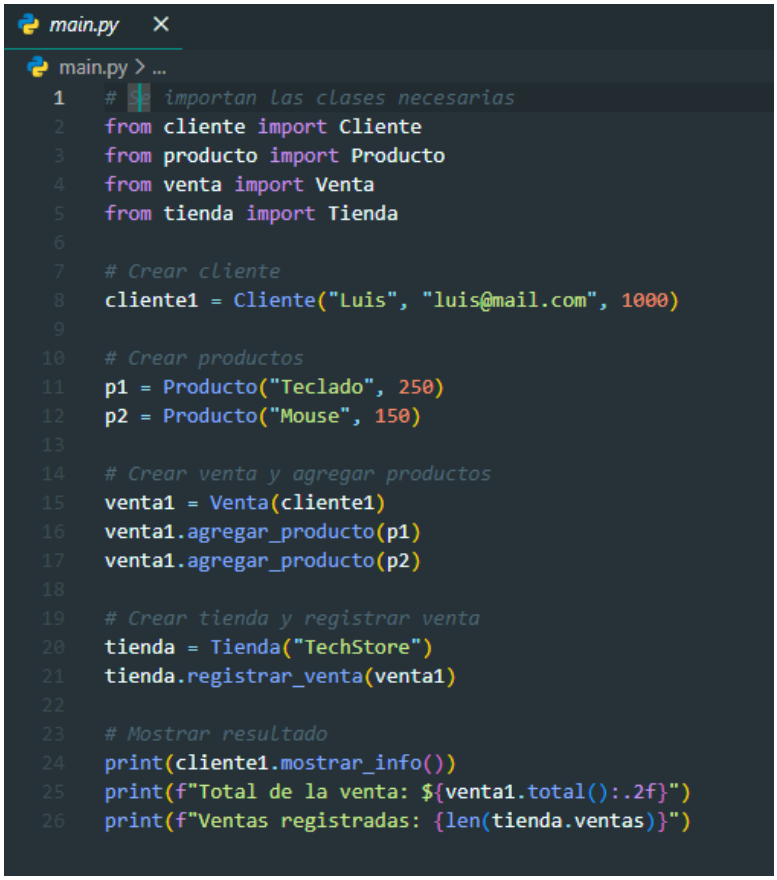
```python
# Se importan las clases necesarias
from usuario import Usuario
# Clase Administrador
class Administrador(Usuario):
    # Inicializador
    def __init__(self, nombre: str, correo: str, permisos: list[str]):
        super().__init__(nombre, correo)
        self.permisos = permisos
    # Método para mostrar información del administrador
    def mostrar_info(self) -> str:
        return f"Administrador: {self.nombre}, Correo: {self.correo}, Permisos: {', '.join(self.permisos)}"
```

- Cliente.py



```python
# Se importan las clases necesarias
from usuario import Usuario
# Clase Cliente
class Cliente(Usuario):
    # Inicializador
    def __init__(self, nombre: str, correo: str, saldo: float):
        super().__init__(nombre, correo)
        self.saldo = saldo
    # Método para mostrar información del cliente
    def mostrar_info(self) -> str:
        return f"Cliente: {self.nombre}, Correo: {self.correo}, Saldo: ${self.saldo:.2f}"
```

- Main.py



```python
# Se importan las clases necesarias
from cliente import Cliente
from producto import Producto
from venta import Venta
from tienda import Tienda

# Crear cliente
cliente1 = Cliente("Luis", "luis@mail.com", 1000)

# Crear productos
p1 = Producto("Teclado", 250)
p2 = Producto("Mouse", 150)

# Crear venta y agregar productos
venta1 = Venta(cliente1)
venta1.agregar_producto(p1)
venta1.agregar_producto(p2)

# Crear tienda y registrar venta
tienda = Tienda("TechStore")
tienda.registrar_venta(venta1)

# Mostrar resultado
print(cliente1.mostrar_info())
print(f"Total de la venta: ${venta1.total():.2f}")
print(f"Ventas registradas: {len(tienda.ventas)}")
```

- Producto.py

producto.py

producto.py > Producto

```python
# Clase Producto
class Producto:
    # Atributo de clase para contar productos
    contador_productos = 0
    # Inicializador
    def __init__(self, nombre: str, precio: float):
        self.nombre = nombre
        self.precio = precio
        # Incrementar el contador de productos al crear una nueva instancia
        Producto.contador_productos += 1
    # Método estático para validar el precio
    @staticmethod
    def es_precio_valido(precio: float) -> bool:
        return precio > 0

    @classmethod
    def total_productos(cls) -> int:
        return cls.contador_productos
```

- Tienda.py

tienda.py

tienda.py > Tienda

```python
# Se importan las clases necesarias
from venta import Venta
# Clase Tienda
class Tienda:
    # Inicializador
    def __init__(self, nombre: str):
        self.nombre = nombre
        self.ventas: list[Venta] = []
    # Método para registrar una venta
    def registrar_venta(self, venta: Venta) -> None:
        self.ventas.append(venta)
```

- Usuario.py

usuario.py

```python
# Se importan las clases necesarias
from abc import ABC, abstractmethod
# Clase Usuario
class Usuario(ABC):
    # Inicializador
    def __init__(self, nombre: str, correo: str):
        self.nombre = nombre
        self.correo = correo
    # Método abstracto para mostrar información del usuario
    @abstractmethod
    def mostrar_info(self) -> str:
        pass
```

- Venta.py

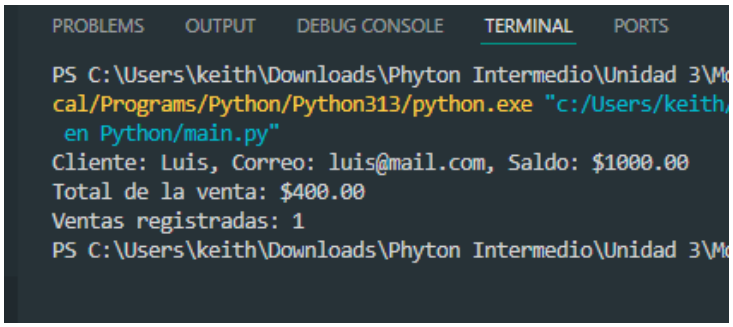
venta.py

```python
# Se importan las clases necesarias
from producto import Producto
from cliente import Cliente
# Clase Venta
class Venta:
    # Inicializador
    def __init__(self, cliente: Cliente):
        self.cliente = cliente
        self.productos: list[Producto] = []
    # Método para agregar productos a la venta
    def agregar_producto(self, producto: Producto) -> None:
        self.productos.append(producto)
    # Método para calcular el total de la venta
    def total(self) -> float:
        return sum(p.precio for p in self.productos)
```

d) Ejecución y pruebas

Ejemplo del funcionamiento del código:

- Resultado:



Reflexiones

- ¿Qué ventajas observaste, al utilizar herencia y clases abstractas en este sistema?
  R= El uso de herencia y clases abstractas me permitió establecer una estructura para los diferentes tipos de usuario, evitando duplicar el código y facilitar la escalabilidad del sistema.

- ¿Por qué es importante separar las clases en archivos distintos, dentro de un sistema modular?
  R= Separar las clases en archivos separados perite una mejor organización del proyecto y esto hace que el código sea más fácil de identificar, leer, depurar y escalar, también permite que se puedan editar diferentes partes del código simultáneamente y poder reutilizar código en proyectos diferentes.

- ¿En qué situaciones prácticas crees que sería útil aplicar composición y agregación, como las usadas en este ejercicio?
  R= El uso de composición es útil cuando un objeto depende totalmente de otro para existir cono facturas que dependen de un catálogo de productos o de un inventario que depende de sus artículos, y la agregación por otro lado se aplica cuando un objeto forma parte de otro, pero puede existir de manera independiente como un carrito de compras de contiene productos.

- ¿Cómo facilitaría la implementación de decoradores de clase, la reutilización de código en un sistema más complejo?
  R= Los decoradores permiten agregar funciones adicionales sin modificar la estructura del código, esto permite reutilizar funciones como validaciones o

registro de actividades, haciendo el código más limpio, modular y fácil de extender.

- ¿Qué mejoras propondrías al sistema, para adaptarlo a un entorno real?
  R= Mejoraría el manejo de la información usando una base de datos para almacenar la información, mejoraría los permisos y roles por tipo de usuario, agregaría una interfaz gráfica, agregaría un sistema de inventario, le agregaría la generación de reportes, una mejor validación para evitar errores.

Conclusiones

El desarrollo de este sistema me permitió aplicar de forma práctica los principios fundamentales de la POO, utilizando los conceptos como herencia, composición, agregación y el hacer uso de los métodos especiales y decoradores, esto a través de la construcción de cada una de la clase y sus relaciones, ayudándome a modelar un sistema de ventas completo, flexible y organizado, demostrando cómo la POO facilita la creación de estructuras de código más limpias y reutilizables.

Otro punto fue la implementación del diagrama UML que contribuyó a visualizar la arquitectura del sistema, permitiéndome ver la importancia del modelado antes de programar, la interacción entre usuarios, productos, ventas y tienda permitió simular un escenario realista donde cada componente cumple una función específica dentro del flujo de trabajo.

En general este ejercicio no solo me ayudo a fortalecer el entendimiento de los pilares de la POO, sino que también evidencia cómo estos conceptos pueden aplicarse para resolver problemas cotidianos de manera eficiente, estructurada y escalable.