PYTHON NIVEL INTERMEDIO

INFOTEC

Centro de investigación e innovación en tecnologías de la información y comunicación

TÍTULO DEL PROJECTO

Modelado y Programación en POO: Herencia, Agregación y Composición

PRESENTA

Armando Gabriel Jacinto López

DOCENTE

Vladimir Morales Pérez

Fecha: 20/11/2025

Introducción

Este informe técnico describe la implementación de un sistema básico de gestión de ventas utilizando los principios fundamentales de la Programación Orientada a Objetos (POO) en Python. El proyecto se centra en la aplicación práctica de **Herencia**, **Composición** y **Agregación** para modelar entidades del mundo real como Usuarios, Clientes, Productos, Ventas y Tiendas.

El objetivo principal es ilustrar cómo la POO facilita la creación de código modular, reutilizable y escalable, estructurando las responsabilidades y las relaciones entre las clases:

1. **Herencia:** Se utiliza para establecer una jerarquía de clases (Clases Cliente y Administrador) que comparten características de una clase base (Usuario), asegurando una interfaz común a través de métodos abstractos.
2. **Composición:** Se aplica en la clase Venta, donde un objeto Venta se construye directamente a partir de un objeto Cliente, siendo el cliente una parte esencial de la venta.
3. **Agregación:** Se aplica en la clase Tienda, que mantiene una colección de objetos Venta. En este caso, la existencia de la venta no depende de la tienda, sino que la tienda "agrega" las ventas a su registro.

Desarrollo

El proyecto se divide en seis archivos, cada uno con una responsabilidad clara, que demuestran los diferentes tipos de relaciones entre objetos.

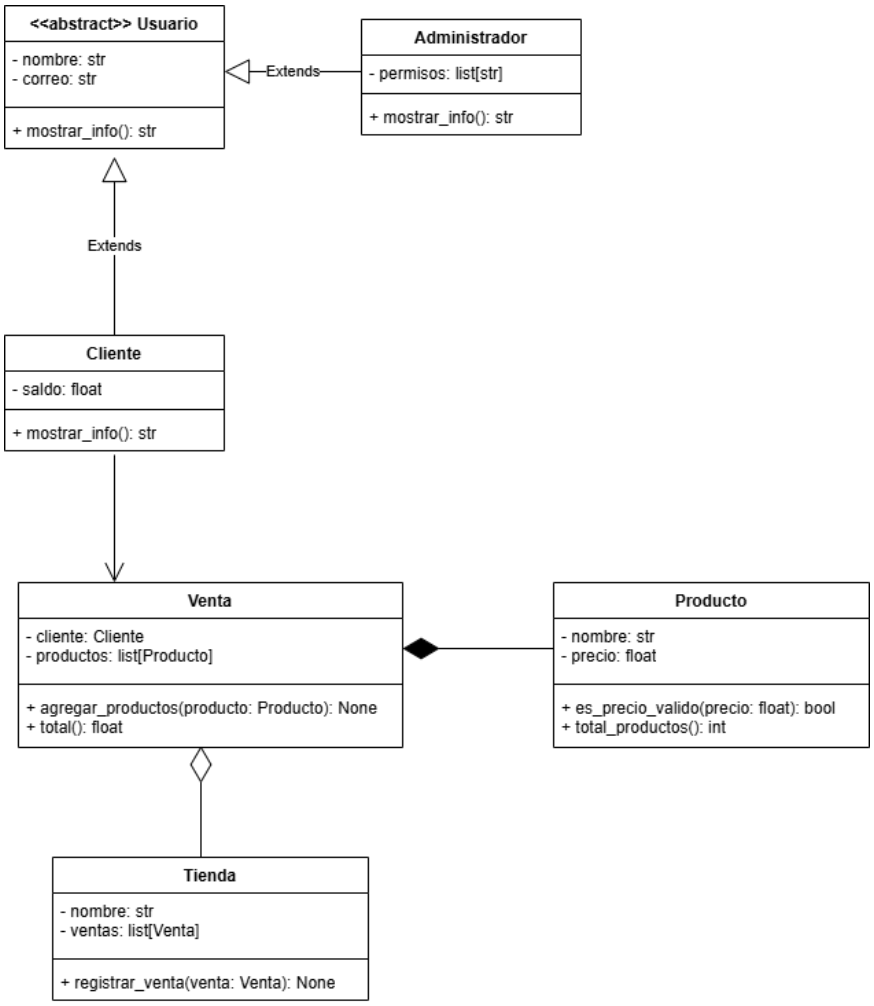

Ilustración 1 Modelado UML, mostrando la relacion de las clases

1. Herencia y Clases Abstractas (usuario.py, cliente.py, administrador.py)

La Herencia se implementa con la clase base Usuario, definida como abstracta utilizando el módulo abc (Abstract Base Classes).

- Clase Usuario (Clase Base Abstracta): Define la estructura común (nombre, correo) y obliga a las subclases a implementar el método

mostrar_info() a través del decorador @abstractmethod. Esto garantiza que cualquier objeto Usuario (o sus derivados) tendrá este comportamiento.

```python
from abc import abstractmethod, ABC

# Clase abstracta, clase base para las demas
class Usuario(ABC):
    def __init__(self, nombre: str, correo: str):
        self.nombre = nombre
        self.correo = correo
    # Metodo que debera ser implementado para las subclases
    @abstractmethod
    def mostrar_info(self) -> str:
        pass
```

Ilustración 2 Clase base abstracta

- Clases Cliente y Administrador (Subclases): Ambas heredan de Usuario y añaden atributos específicos (saldo para Cliente y permisos para Administrador), además de implementar su propia versión del método mostrar_info().

```python
from usuario import Usuario
# Subclase de la clase Usuario
class Administrador(Usuario):
    def __init__(self, nombre: str, correo: str, permisos: list[str]):
        super().__init__(nombre, correo)
        self.permisos = permisos

    def mostrar_info(self) -> str:
        return f'Administrador: {self.nombre}, Correo: {self.correo}, Permisos: {', '.join(self.permisos)}'
```

Ilustración 3 Subclase Administrador

```python
from usuario import Usuario
# Subclase de la clase Usuario
class Cliente(Usuario):
    def __init__(self, nombre: str, correo: str, saldo: float):
        super().__init__(nombre, correo)
        self.saldo = saldo

    def mostrar_info(self) -> str:
        return f'Cliente: {self.nombre}\nCorreo: {self.correo}\nSaldo: ${self.saldo:.2f}'
```

Ilustración 4 Subclase Cliente

2. Clases Estándar y Métodos Especiales (producto.py)

La clase Producto ejemplifica el uso de atributos y métodos de clase:

- Atributo de Clase: contador_productos es compartido por todas las instancias y se incrementa en el constructor (__init__) para rastrear el número total de productos creados.
- Método de Clase (@classmethod): total_productos() accede y devuelve el atributo de clase contador_productos.
- Método Estático (@staticmethod): es_precio_valido() realiza una verificación de lógica sin necesidad de acceder a los datos de la instancia (self) o de la clase (cls), por lo que no requiere ningún argumento.

```python
class Producto():
    # Datos compartidos atraves de todas las instancias
    contador_productos = 0
    # self, indica la instancia
    def __init__(self, nombre: str, precio: float):
        self.nombre = nombre
        self.precio = precio
        Producto.contador_productos += 1
    # logicamente pertenece a la clase(no requiere acceso a la clase o instancia)
    @staticmethod
    def es_precio_valido(precio: float) -> bool:
        return precio > 0 and isinstance(precio, float)
    # acceso a la clase
    @classmethod
    def total_productos(cls) -> int:
        return cls.contador_productos
```

Ilustración 5 Aplicación de decoradores en clases

3. Composición y Agregación (venta.py y tienda.py)

Estas clases modelan las relaciones más complejas:

- Composición (Venta): La clase Venta contiene una referencia directa a un objeto Cliente (self.cliente). Una venta se compone de un cliente y una lista de productos. Sin un cliente, la venta carece de sentido.

- Agregación (Tienda): La clase Tienda mantiene una lista de objetos Venta (self.ventas). Esta es una relación de Agregación porque la existencia de una Venta (el objeto creado en el entorno) es independiente de que la Tienda la registre o no.

```python
from producto import Producto
from cliente import Cliente
# Relacion con la clase Cliente, y Producto
class Venta():
    def __init__(self, cliente: Cliente):
        self.cliente = cliente
        self.productos: list[Producto] = []

    def agregar_productos(self, producto: Producto) -> None:
        self.productos.append(producto)

    def total(self) -> float:
        return sum(p.precio for p in self.productos)
```

Ilustración 6 Clase Venta, composición con la clase Cliente

```python
from venta import Venta
# Relacion con la clase Venta
class Tienda:
    def __init__(self, nombre: str):
        self.nombre = nombre
        self.ventas: list[Venta] = []

    def registrar_venta(self,venta: Venta) -> None:
        self.ventas.append(venta)
```

Ilustración 7 Clase Tienda, agregación con la clase Venta

4. Flujo de Ejecución (main.py)

El archivo principal orquesta las interacciones: se crean instancias de las clases más independientes (Cliente, Producto), se utilizan para crear la Venta (Composición), y finalmente, la Venta se registra en la Tienda (Agregación), demostrando la funcionalidad completa del modelo.

```python
from cliente import Cliente
from producto import Producto
from venta import Venta
from tienda import Tienda

# Crear el cliente
cliente_1 = Cliente('Luis', 'luis@gmail.com', 1000

# Crear productos
producto_1 = Producto('Teclado', 250)
producto_2 = Producto('Mouse', 150)

# Crear venta y agregar productos
venta_1 = Venta(cliente_1)
venta_1.agregar_productos(producto_1)
venta_1.agregar_productos(producto_2)

# Crear tienda y registrar venta
tienda = Tienda('TechStore')
tienda.registrar_venta(venta_1)

# Mostrar resultado
print(cliente_1.mostrar_info())
print(f'Total de la venta: ${venta_1.total()}')
print(f'Ventas registradas: {len(tienda.ventas)}')
```

Ilustración 8 Archivo principal

```
Cliente: Luis
Correo: luis@gmail.com
Saldo: $1000.00
Total de la venta: $400
Ventas registradas: 1
```

Ilustración 9 Resultados de la ejecución de main.py

Conclusión

El proyecto demostró la capacidad de modelar un sistema transaccional utilizando los principios de la POO, lo que resulta en un código altamente estructurado y fácil de mantener.

- Reutilización y Uniformidad (Herencia): La clase Usuario aseguró que tanto Cliente como Administrador tuvieran una estructura base y un método mostrar_info garantizado, promoviendo la consistencia en el diseño.
- Modelado de Relaciones (Composición y Agregación): La distinción clara entre la Composición (donde el Cliente es una parte fundamental de la Venta) y la Agregación (donde la Tienda simplemente registra objetos Venta independientes) permitió un modelado preciso del dominio.

En última instancia, el uso de la POO facilita la gestión de la complejidad del sistema, preparando el código para futuras expansiones, como la adición de nuevos tipos de usuarios (por ejemplo, Proveedor) o la implementación de nuevas lógicas de negocio (por ejemplo, gestión de inventario).