INFORME PROGRAMACIÓN

HERENCIA

Integrantes:

- Anderstone Paccha
- RodrigoSeraquive
- Patricio Poma

Segundo ciclo
13/06/2025

Humano

Estratégico

UN SITIO GENIAL

1. Introducción

La herencia es un pilar fundamental de la programación orientada a objetos, ya que permite reutilizar código y establecer relaciones jerárquicas entre clases. Este informe presenta la aplicación práctica de la herencia en distintos contextos, como zoológicos, universidades, tiendas, bancos y hoteles, mostrando cómo una clase base puede compartir atributos y métodos comunes con sus subclases, facilitando el diseño y mantenimiento de sistemas más organizados y eficientes.

2. Objetivo General

Aplicar el concepto de herencia en la programación orientada a objetos para resolver distintos problemas del mundo real, promoviendo la reutilización de código y la correcta estructuración de clases.

3. Resolución de problemas

PROBLEMA #1

Sistema de Gestión de Animales en un Zoológico

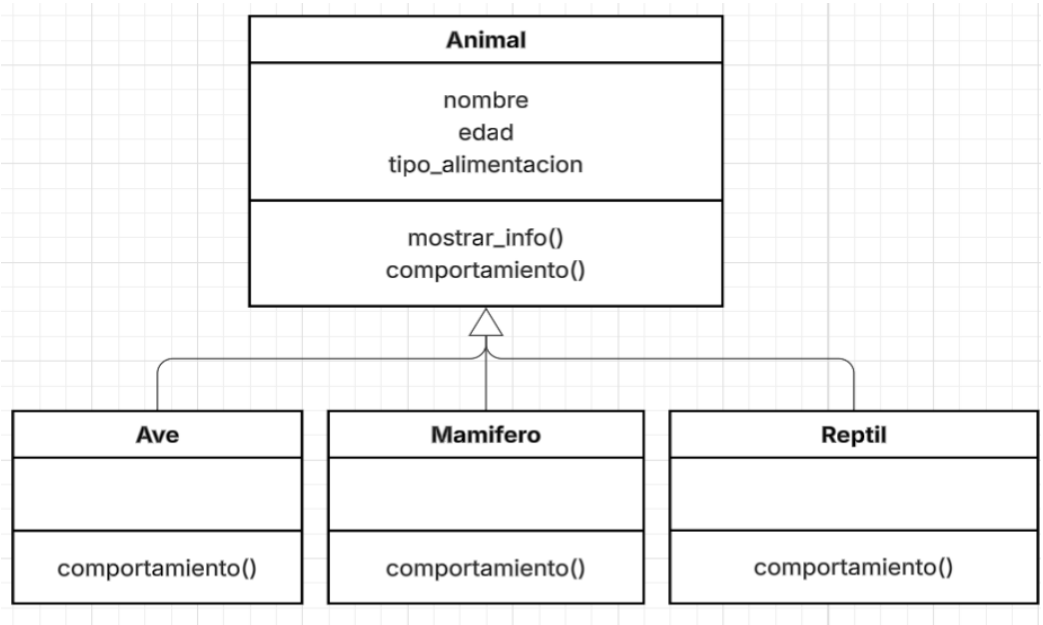
Problema:

Un zoológico necesita un sistema para gestionar distintos tipos de animales. Todos los animales comparten nombre, edad y tipo de alimentación, pero algunos son aves, otros mamíferos, y otros reptiles, con comportamientos específicos (como volar o reptar).

Objetivo:

- Diseñar clases que modelen correctamente las relaciones.
- Permitir imprimir información y comportamiento específico de cada tipo de animal.

✓ **DIAGRAMA UML**



✓ **CÓDIGO**

SistemaBancario.py | GestionAcademica.py | zoologico.py | SistemaFacturacion.py

zoologico.py > Reptil

```python
class Animal:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, edad, tipoAlimentacion):
        self.nombre = nombre
        self.edad = edad
        self.tipoAlimentacion = tipoAlimentacion

    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        print(f"Nombre: {self.nombre}")
        print(f"Edad: {self.edad}")
        print(f"Tipo de Alimentación: {self.tipoAlimentacion}")

    Tabnine | Edit | Test | Explain | Document
    def comportamiento(self):
        print("Este animal tiene un comportamiento normal.")

# Clase Ave hereda de Animal y redefine el comportamiento
class Ave(Animal):
    Tabnine | Edit | Test | Explain | Document
    def comportamiento(self):
        print(f"{self.nombre} está volando.")

# Clase Mamifero hereda de Animal y redefine el comportamiento
class Mamifero(Animal):
    Tabnine | Edit | Test | Explain | Document
    def comportamiento(self):
        print(f"{self.nombre} está caminando.")

# Clase Reptil hereda de Animal y redefine el comportamiento
class Reptil(Animal):
    Tabnine | Edit | Test | Explain | Document
    def comportamiento(self):
        print(f"{self.nombre} está reptando.")

# Se crean objetos de cada tipo de animal
loro = Ave("Loro", 2, "Herbívoro")
tigre = Mamifero("Tigre", 5, "Carnívoro")
iguana = Reptil("Iguana", 3, "Omnívoro")

# Se recorren los animales mostrando su información y comportamiento
for animal in [loro, tigre, iguana]:
    animal.mostrarInfo()
    animal.comportamiento()
    print("-" * 30)
```

✓ **Ejecución:**

```
PS C:\Users\DELL\Downloads\POO> python zoologico.py
Nombre: Loro
Edad:2
Tipo de Alimentación: Herbivoro
Loro esta volando.
------------------------------
Nombre: Tigre
Edad:5
Tipo de Alimentación: Carnivoro
Tigre esta caminando.
------------------------------
Nombre: Iguana
Edad:3
Tipo de Alimentación: Omnivoro
Iguana esta raptando
------------------------------
```

USO DE HERENCIA

Se utiliza una clase base llamada `Animal` que contiene atributos comunes como `nombre`, `edad` y `tipo_alimentacion`. Luego, se crean subclases como `Ave`, `Mamifero` y `Reptil` que heredan de `Animal` y añaden comportamientos específicos como `volar()` o `reptar()`. La **herencia** se implementa en las líneas donde se define cada subclase, por ejemplo: class Ave(Animal): Aquí, `Ave` hereda de `Animal`, reutilizando sus atributos y métodos y agregando nuevos.

PROBLEMA #2

Gestión Académica en una Universidad

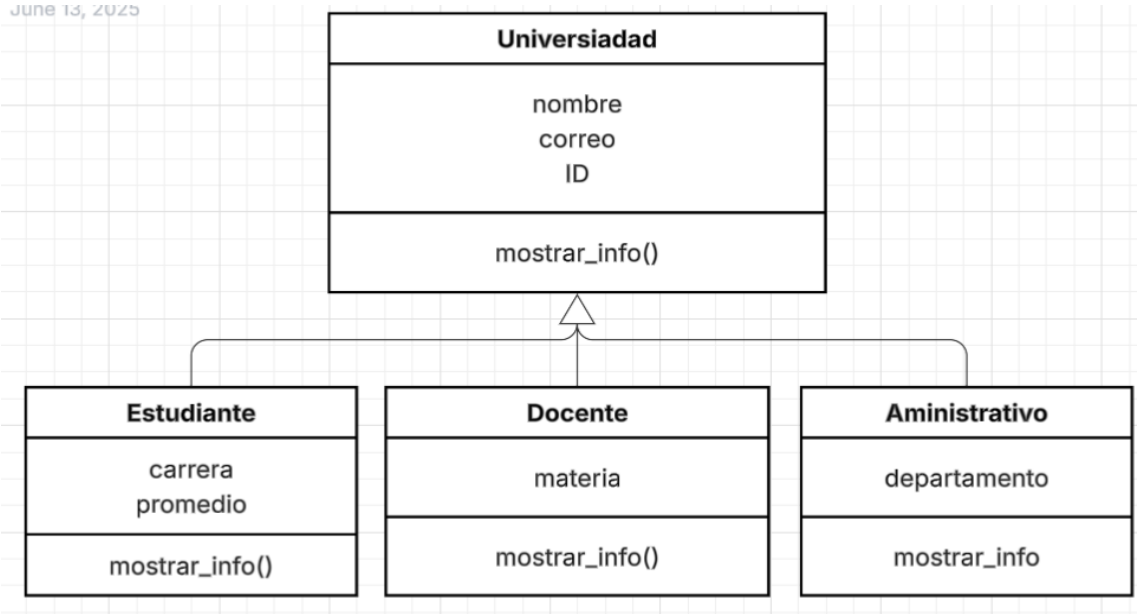
Problema:

Una universidad desea llevar el control de su comunidad. Todos los miembros (estudiantes, docentes, administrativos) tienen nombre, ID y correo. Los estudiantes tienen carrera y promedio; los docentes tienen materia que imparten; y los administrativos, departamento al que pertenecen.

Objetivo:

- Aplicar herencia para evitar repetir código.
- Mostrar los datos de cada miembro con métodos personalizados.

✓ **DIAGRAMA UML**



✓ **CÓDIGO**

ReservasHotel.py | SistemaFacturacion.py | GestionAcademica.py | zoologico.py | SistemaBancario.py

C: > Users > DELL > Downloads > POO > GestionAcademica.py > ...

```python
class Universidad:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, id, correo):
        self.nombre = nombre
        self.id = id
        self.correo = correo
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        print(f"Nombre: {self.nombre}")
        print(f"ID: {self.id}")
        print(f"Correo: {self.correo}")
# Clase Estudiante hereda de Universidad y agrega 'carrera' y 'promedio'
class Estudiante(Universidad):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, id, correo, carrera, promedio):
        super().__init__(nombre, id, correo)
        self.carrera = carrera
        self.promedio = promedio
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        super().mostrarInfo()
        print(f"Carrera: {self.carrera}")
        print(f"Promedio: {self.promedio}")
# Clase Docente hereda de Universidad y agrega 'materia'
class Docente(Universidad):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, id, correo, materia):
        super().__init__(nombre, id, correo)
        self.materia = materia
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        super().mostrarInfo()
        print(f"Materia que imparte: {self.materia}")
# Clase Administrativo hereda de Universidad y agrega 'departamento'
class Administrativo(Universidad):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, id, correo, departamento):
        super().__init__(nombre, id, correo)
        self.departamento = departamento
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        super().mostrarInfo()
        print(f"Departamento: {self.departamento}")
# Se crean objetos para cada tipo de miembro de la universidad
Estudiante1 = Estudiante("Carla", "EST123", "carla@utpl.edu.ec", "Medicina", 9.2)
docente1 = Docente("Ramiro", "MGI321", "ramirez@utpl.edu.ec", "POO")
admin1 = Administrativo("Marco", "ADM789", "marco@utpl.edu.ec", "Ingenierías")
# Lista que almacena todos los miembros
miembros = [Estudiante1, docente1, admin1]
# Se recorre la lista mostrando la información completa de cada persona
for persona in miembros:
    print("-" * 30)
    persona.mostrarInfo()
```

✓ **Ejecución:**

```
PS C:\Users\DELL\Downloads\POO> python GestionAcademica.py
------------------------------
Nombre: Carla
ID: EST123
Correo: carla@utpl.edu.ec
Carrera: Medicina
Promedio: 9.2
------------------------------
Nombre: Ramiro
ID: MGI321
Correo: ramirez@utpl.edu.ec
Materia que imparte: POO
------------------------------
Nombre: Marco
ID: ADM789
Correo: marco@utpl.edu.ec
Departamento: Ingenierías
PS C:\Users\DELL\Downloads\POO>
```

USO DE HERENCIA

Se define una clase base `Universidad` con los atributos comunes: `nombre`, `id` y `correo`. Las subclases `Estudiante`, `Docente` y `Administrativo` heredan de esta clase para evitar repetir código y agregan sus propios atributos como `carrera`, `materia`, o `departamento`. La herencia se aplica así: class Estudiante(Universidad): Esta línea indica que `Estudiante` hereda todos los atributos y métodos de `Universidad`.

PROBLEMA #3

Sistema de Facturación de Tienda de Electrónica

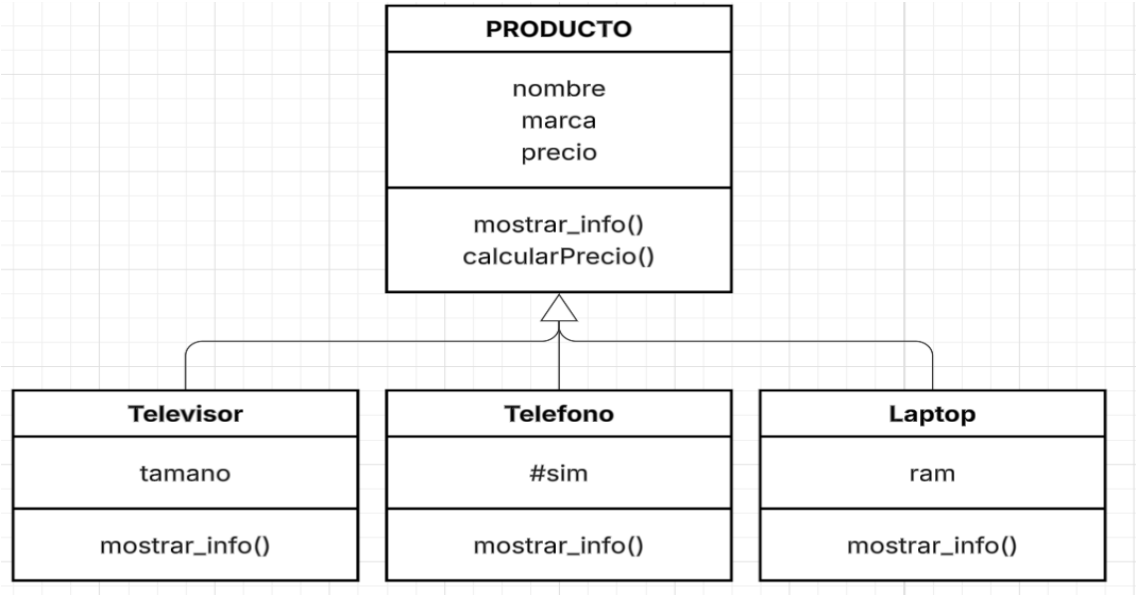
Problema:

Una tienda necesita facturar productos electrónicos. Todos los productos tienen nombre y precio. Algunos productos como televisores tienen tamaño en pulgadas, los teléfonos tienen número de SIM, y las laptops tienen memoria RAM.

Objetivo:

- Usar herencia para representar la variedad de productos.
- Calcular el precio final con IVA usando un método común.

✓ **DIAGRAMA UML**



✓ **CÓDIGO**

SistemaBancario.py | GestionAcademica.py | zoologico.py | SistemaFacturacion.py

SistemaFacturacion.py > ...

```python
class Producto:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, marca, precio):
        self.nombre = nombre
        self.precio = precio
        self.marca = marca
    Tabnine | Edit | Test | Explain | Document
    def calcularPrecio(self):
        iva = 0.15  # IVA del 15%
        return self.precio * (1 + iva)
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        print(f"Producto: {self.nombre}")
        print(f"Marca: {self.marca}")
        print(f"Precio sin IVA: ${self.precio:.2f}")
# Clase Televisor hereda de Producto y agrega atributo 'tamaño'
class Televisor(Producto):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, marca, precio, tamaño):
        super().__init__(nombre, marca, precio)
        self.tamaño = tamaño  # Nuevo atributo exclusivo
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        super().mostrarInfo()
        print(f"Tamaño: {self.tamaño} pulgadas")
# Clase Telefono hereda de Producto y agrega atributo 'sim'
class Telefono(Producto):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, marca, precio, sim):
        super().__init__(nombre, marca, precio)
        self.sim = sim  # Nuevo atributo exclusivo
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        super().mostrarInfo()
        print(f"Número de SIM: {self.sim}")
# Clase Laptop hereda de Producto y agrega atributo 'ram'
class Laptop(Producto):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, nombre, marca, precio, ram):
        super().__init__(nombre, marca, precio)
        self.ram = ram  # Nuevo atributo exclusivo
    Tabnine | Edit | Test | Explain | Document
    def mostrarInfo(self):
        super().mostrarInfo()
        print(f"Memoria RAM: {self.ram} GB")
# Se crean objetos de cada clase hija con sus respectivos atributos
tv = Televisor("TELEVISOR", "LG", 550, 55)
celular = Telefono("CELULAR", "SAMSUNG", 200, 2)
laptop = Laptop("LAPTOP", "ASUS", 700, 8)
# Lista con los productos creados
productos = [tv, celular, laptop]
# Se recorre la lista y se muestra la información de cada producto con su precio con IVA
for producto in productos:
    print("-" * 30)
    producto.mostrarInfo()
    print(f"Precio con IVA: ${producto.calcularPrecio():.2f}")
```

✓ **Ejecución:**

```
PS C:\Users\DELL\Downloads\POO> python SistemaFacturacion.py
------------------------------
Producto: TELEVISOR
Marca: LG
Precio sin IVA: $550.00
Tamaño: 55 pulgadas
Precio con IVA: $632.50
------------------------------
Producto: CELULAR
Marca: SAMSUNG
Precio sin IVA: $200.00
Número de SIM: 2
Precio con IVA: $230.00
------------------------------
Producto: LAPTOP
Marca: ASUS
Precio sin IVA: $700.00
Memoria RAM: 8 GB
Precio con IVA: $805.00
```

USO DE HERENCIA

Se usa una clase padre `Producto` con atributos `nombre` y `precio`, y un método común para calcular el precio con IVA. Las subclases `Televisor`, `Telefono` y `Laptop` heredan de `Producto` y añaden sus propios atributos como `tamano`, `num_sim`, y `memoria_ram`. La **herencia** se implementa como sigue: class Televisor(Producto): Esto permite que todas las subclases compartan el método de cálculo del precio final, sin repetirlo.

PROBLEMA #4

Sistema Bancario

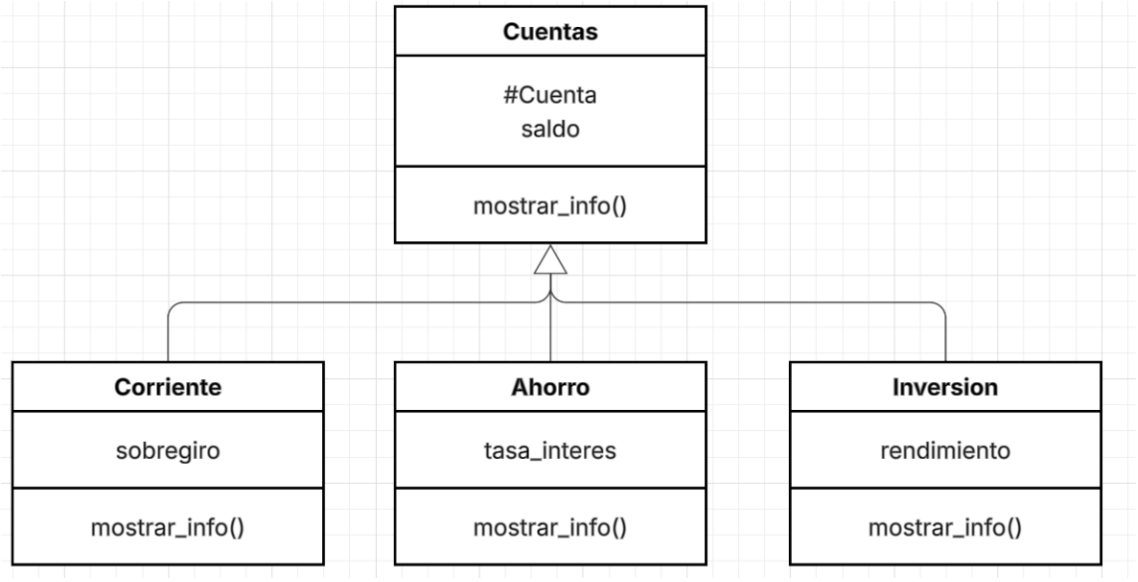
Problema:

Un banco ofrece distintos tipos de cuentas: cuentas de ahorro, cuentas corriente y cuentas de inversión. Todas las cuentas tienen número de cuenta y saldo. Las cuentas de ahorro tienen una tasa de interés; las corrientes permiten sobregiro; las de inversión generan un rendimiento.

Objetivo:

- Aplicar herencia para evitar repetir código.
- Mostrar los datos de cada miembro con métodos personalizados.

✓ **DIAGRAMA UML**



✓ **CÓDIGO**

SistemaBancario.py | ReservasHotel.py | GestionAcademica.py | zoologico.py | SistemaFacturacion.py

SistemaBancario.py > ...

```python
class Cuentas:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_cuenta, saldo):
        self.num_cuenta = num_cuenta
        self.saldo = saldo
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        print(f"Número de Cuenta: {self.num_cuenta}")
        print(f"Saldo: {self.saldo}")
# Clase Corriente hereda de Cuentas y agrega el atributo 'sobregiro'
class Corriente(Cuentas):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_cuenta, saldo, sobregiro):
        super().__init__(num_cuenta, saldo)
        self.sobregiro = sobregiro
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        super().mostrar_info()
        print(f"Sobregiros: {self.sobregiro}")
        print('-' * 30)
# Clase Ahorro hereda de Cuentas y agrega 'tasa_interes'
class Ahorro(Cuentas):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_cuenta, saldo, tasa_interes):
        super().__init__(num_cuenta, saldo)
        self.tasa_interes = tasa_interes
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        super().mostrar_info()
        print(f"Tasa de interés: {self.tasa_interes}")
        print('-' * 30)
# Clase Inversion hereda de Cuentas y agrega 'rendimiento'
class Inversion(Cuentas):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_cuenta, saldo, rendimiento):
        super().__init__(num_cuenta, saldo)
        self.rendimiento = rendimiento
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        super().mostrar_info()
        print(f"Rendimiento: {self.rendimiento}")
        print('-' * 30)
# Lista con distintos tipos de cuentas
banco = [
    Corriente(2901482942, 200000, 500),
    Ahorro(2485029420, 5430000, '10%'),
    Inversion(5293502395, 99999999, 210000)
]
# Muestra información de todas las cuentas en el sistema bancario
print("Información del Sistema Bancario:")
print('-' * 30)
for cuenta in banco:
    cuenta.mostrar_info()

```

✓ **Ejecución:**

```
PS C:\Users\DELL\Downloads\POO> python SistemaBancario.py
Información del Sistema Bancario:
------------------------------
Número de Cuenta: 2901482942
Saldo: 200000
Sobregiros: 500
------------------------------
Número de Cuenta: 2485029420
Saldo: 5430000
Tasa de interés: 10%
------------------------------
Número de Cuenta: 5293502395
Saldo: 99999999
Rendimiento: 210000
------------------------------
```

USO DE HERENCIA

Una clase base `CuentaBancaria` almacena atributos comunes: `numero_cuenta` y `saldo`. Las subclases `CuentaAhorro`, `CuentaCorriente` y `CuentaInversion` heredan de esta clase y definen atributos o métodos específicos como `tasa_interes`, `sobregiro`, o `rendimiento`. Ejemplo de línea con herencia: class CuentaAhorro(CuentaBancaria): Esto permite evitar duplicación de atributos y aprovechar métodos compartidos.

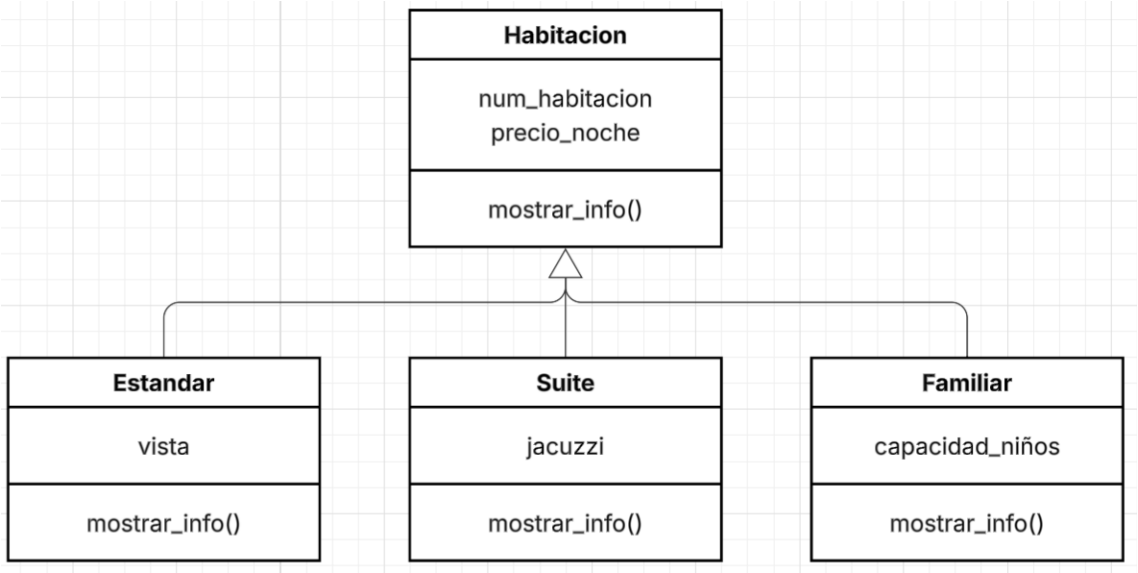
PROBLEMA #5

Sistema de Reservas de Hotel

Problema:

Un hotel permite reservar habitaciones estándar, suites y habitaciones familiares. Todas tienen número de habitación y precio por noche. Las suites tienen jacuzzi, las familiares capacidad para niños, y el estándar, si tienen vista o no.

- ✓ **DIAGRAMA UML**



✓ **CÓDIGO**

SistemaBancario.py | ReservasHotel.py | GestionAcademica.py | zoologico.py | SistemaFacturacion.py

ReservasHotel.py > ...

```python
class Hotel:
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_habitacion, precio_noche):
        self.num_habitacion = num_habitacion
        self.precio_noche = precio_noche
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        print(f"Número de habitación: {self.num_habitacion}")
        print(f"Precio por noche: {self.precio_noche}")
# Clase Suites hereda de Hotel y agrega el atributo 'jacuzzi'
class Suites(Hotel):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_habitacion, precio_noche, jacuzzi):
        super().__init__(num_habitacion, precio_noche)
        self.jacuzzi = jacuzzi
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        super().mostrar_info()
        print(f"Tiene Jacuzzi?: {self.jacuzzi}")
        print('-' * 40)
# Clase Familiares hereda de Hotel y agrega 'capacidad_niños'
class Familiares(Hotel):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_habitacion, precio_noche, capacidad_ninos):
        super().__init__(num_habitacion, precio_noche)
        self.capacidad_ninos = capacidad_ninos
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        super().mostrar_info()
        print(f"Capacidad para niños: {self.capacidad_ninos}")
        print('-' * 40)
# Clase Estandar hereda de Hotel y agrega 'vista'
class Estandar(Hotel):
    Tabnine | Edit | Test | Explain | Document
    def __init__(self, num_habitacion, precio_noche, vista):
        super().__init__(num_habitacion, precio_noche)
        self.vista = vista
    Tabnine | Edit | Test | Explain | Document
    def mostrar_info(self):
        super().mostrar_info()
        print(f"Tienen vista?: {self.vista}")
        print('-' * 40)
# Lista con diferentes tipos de habitaciones creadas
habitaciones = [
    Suites(510, 380, 'Sí'),
    Familiares(302, 220, 4),
    Estandar(95, 100, 'Sí')
]
print("Información del Sistema de Reservas de Hotel:")
print('-' * 40)
# Se recorre cada habitación para mostrar su información
for habitacion in habitaciones:
    habitacion.mostrar_info()

```

✓ **Ejecución:**

```
PS C:\Users\DELL\Downloads\POO>  python ReservasHotel.py
Información del Sistema de Reservas de Hotel:
----------------------------------------
Número de habitación: 510
Precio por noche: 380
Tiene Jacuzzi?: Sí
----------------------------------------
Número de habitación: 302
Precio por noche: 220
Capacidad para niños: 4
----------------------------------------
Número de habitación: 95
Precio por noche: 100
Tienen vista?: Sí
----------------------------------------
```

USO DE HERENCIA

Se crea una clase `Habitacion` con los atributos comunes `numero_habitacion` y `precio_noche`. Luego, `HabitacionEstandar`, `Suite`, y `HabitacionFamiliar` heredan de `Habitacion` y añaden sus propiedades específicas como `vista`, `jacuzzi`, o `capacidad_ninos`. Se usa herencia así: class Suite(Habitacion): Con esto, se reutiliza código común y se extiende según el tipo de habitación.

4. Conclusiones:

- La herencia permite reutilizar atributos y métodos comunes entre clases, reduciendo la redundancia de código.
- La implementación de subclases facilita la especialización de objetos según sus características particulares.
- El uso de la POO permite modelar sistemas del mundo real de manera intuitiva y estructurada.
- Los diagramas UML fueron fundamentales para planificar y visualizar la relación entre las clases antes de codificar.