Programación 2

Herencia y Polimorfismo

Repaso

La programación orientada a objetos propone identificar durante el desarrollo de requerimientos los **objetos** del problema, caracterizarlos a través de sus principales **atributos** y agruparlos en **clases**.

En la etapa de diseño del sistema se completa la especificación de las clases modelando el **comportamiento** de los objetos.

El resultado de un **diseño orientado a objetos** incluye un **diagrama de clases** escrito en un **lenguaje de modelado**.

El diagrama especifica la **colección** de clases y sus **relaciones**.

Repaso

Durante la implementación del sistema se escribe el **código** de cada clase en un lenguaje de programación, manteniendo las relaciones especificadas en el diagrama de clases.

En ejecución cada uno de los **objetos del problema** quedará asociado a un **objeto de software**. Cada objeto de software tiene un **estado interno** y cuando recibe un mensaje actúa de acuerdo al **comportamiento** definido por su **clase**.

Una clase puede pensarse como un **tipo de dato** a partir del cual se declaran variables y se crean objetos.

Herencia

El proceso de clasificación realizado en un diseño orientado a objetos se organiza en niveles.

En el primer nivel los objetos se agrupan en **clases** de acuerdo a sus atributos y comportamientos.

En el segundo nivel del proceso de clasificación las clases se estructuran a través de un mecanismo de **especialización-generalización** llamado **herencia**.

La herencia favorece la **reusabilidad** y la **extensibilidad** del software.

Herencia - Conceptos clave

- **Clase Base o Padre**: Es la clase de la que otras clases heredan. Contiene atributos y métodos comunes que pueden ser compartidos.
- **Clase Derivada o Hija**: Es la clase que hereda de la clase base. Puede usar los atributos y métodos de la clase base y además añadir o modificar comportamientos.
- **Reutilización de Código**: Las subclases pueden reutilizar las definiciones de la clase base y extenderlas o sobrescribirlas.
- **Polimorfismo**: Una clase hija puede implementar métodos de la clase base con su propia versión (sobreescritura).

Caso de estudio: juego de rol simple

En este juego de rol, existen Personajes que tienen atributos como nombre, vida, ataque y defensa, pueden equiparse con un arma y también pueden abrir cajas sorpresa que aleatoriamente modifican sus atributos y atacarse entre sí.

Existen dos tipos específicos de personajes:

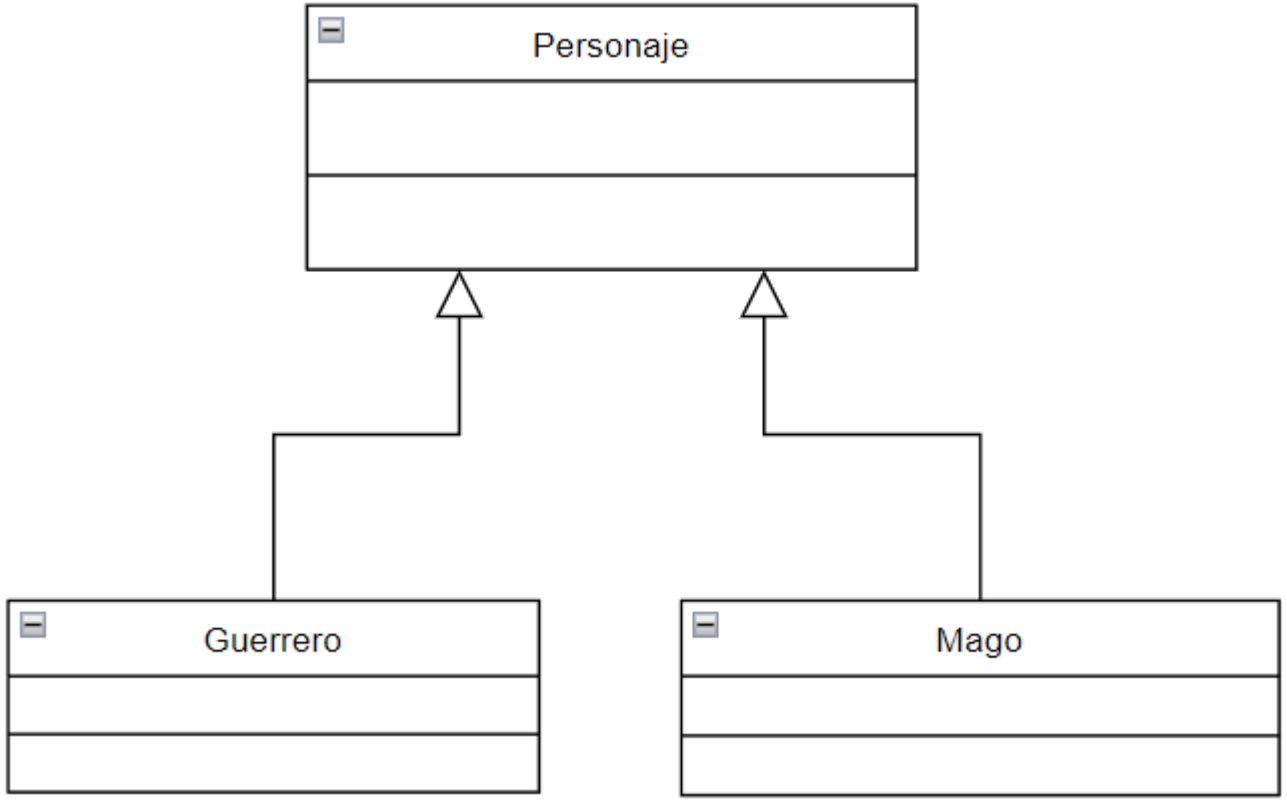
- Guerrero: Es un tipo de personaje que cuando es atacado reduce el daño recibido a la mitad gracias a su entrenamiento, además de una habilidad especial llamada grito de guerra, que le otorga un aumento temporal de ataque (+15p. válidos para un ataque) y le consume 1 punto de vida.
- Mago: Es un tipo de personaje que tiene un atributo adicional llamado mana, que le permite lanzar hechizos para atacar a sus enemigos con un 30% más de su valor de ataque, consumiendo 10 puntos de su mana. Además cuando un mago recibe un ataque puede aleatoriamente repelerlo consumiendo 5 puntos de mana.

Ambos personajes pueden realizar las acciones generales de los personajes, como atacar y abrir cajas sorpresa, pero también tienen habilidades propias como grito de guerra (Guerrero) y lanzar hechizo (Mago).

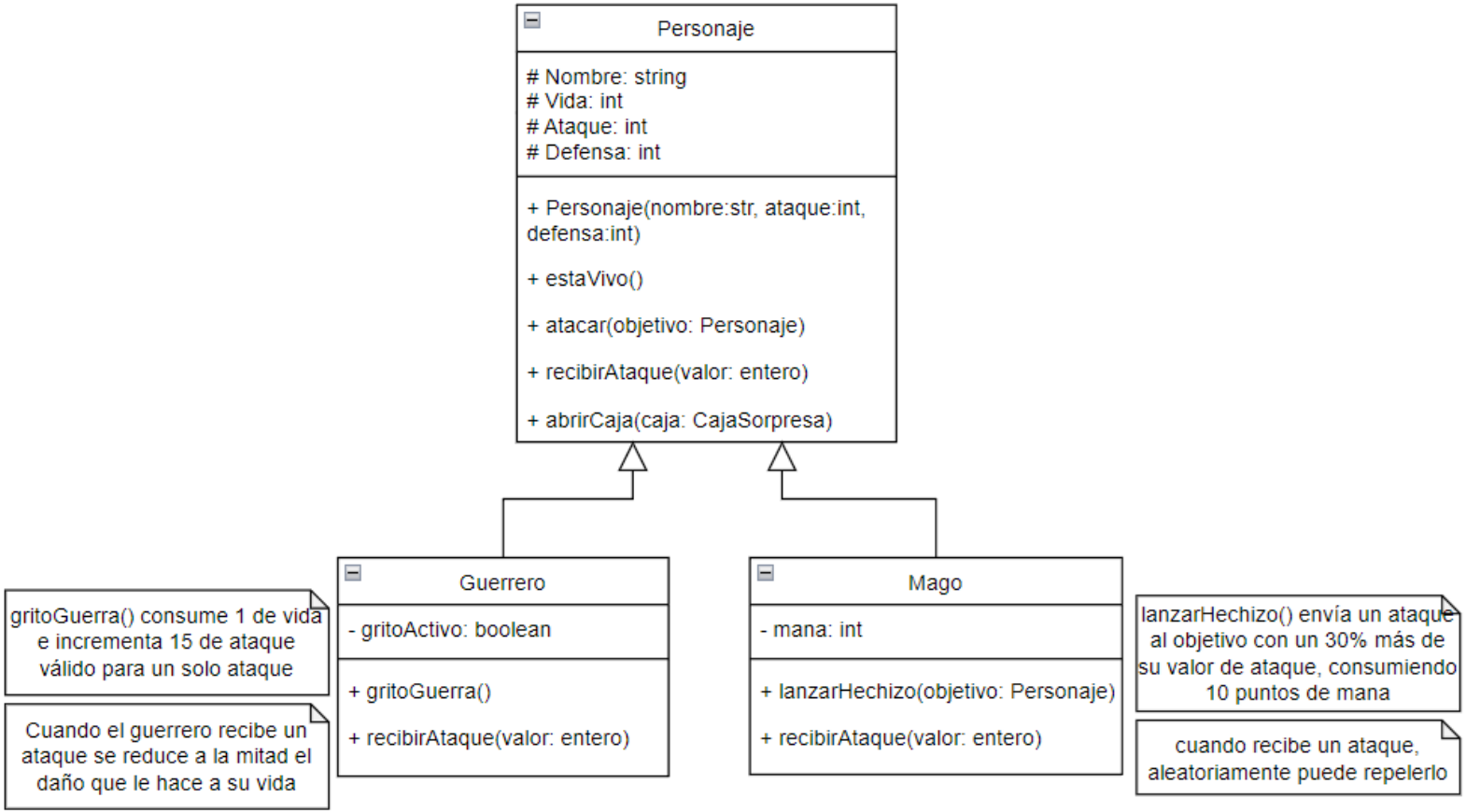
Un Personaje puede equiparse con un Arma, la cual define el daño adicional que puede infligir en un ataque. Las cajas sorpresa ofrecen un valor aleatorio para mejorar o empeorar los atributos de los personajes (vida, ataque, defensa).

Caso de estudio: juego de rol simple

Por lo que detalla el enunciado sabemos que los personajes tienen atributos y comportamientos comunes, entonces podemos agruparlos en la clase personaje, y luego en las subclases definir los atributos y comportamientos específicos



Caso de estudio: juego de rol simple



Caso de estudio: juego de rol simple

Tanto la clase Guerrero como la clase Mago se vinculan con la clase Personaje a través de un relación de herencia.

La clase Guerrero (al igual que la clase Mago) especializa a la clase Personaje.

La clase Guerrero (al igual que la clase Mago) es una subclase o clase derivada de la clase base Personaje.

En python la clase más general es Object.

Todas las clases que implementamos heredan implícitamente de Object.

Herencia en python - definición de la herencia

La herencia en Python permite que una clase (subclase) herede los atributos y métodos de otra clase (superclase). Para implementarla correctamente y aprovechar al máximo su flexibilidad, es importante considerar varios aspectos relacionados con la visibilidad de los atributos y la forma de sobreescribir métodos.

```
class SuperClase:
    # atributos y métodos de la clase base
    pass

class SubClase(SuperClase):
    # atributos y métodos de la subclase
    pass
```

Herencia en python - uso de super()

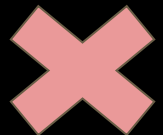
Para asegurarte de que los atributos de la clase base se inicialicen correctamente en una subclase, utiliza `super()` en el constructor de la subclase. Esto permite llamar al constructor de la clase base y heredar sus características:

```python
class SuperClase:
    # atributos y métodos de la clase base
    def __init__(self, nombre):
        self._nombre = nombre

class SubClase(SuperClase):
    # atributos y métodos de la subclase
    def __init__(self, nombre):
        super().__init__(nombre)
```

Herencia en python - atributos protegidos

En Python, los atributos que comienzan con doble guion bajo (__) se consideran privados y no son accesibles directamente desde las subclases. Si deseamos que los atributos de la clase base sean accesibles y modificables por las subclases, es recomendable usar atributos protegidos, que comienzan con un solo guion bajo (_).

```python
class SuperClase:
    # atributos y métodos de la clase base PRIVADOS
    def __init__(self, nombre):
        self.__nombre = nombre

class SubClase(SuperClase):
    # atributos y métodos de la subclase
    def __init__(self, nombre):
        super().__init__(nombre)

    def modificarNombre(self, nuevoNombre):
        self.__nombre = nuevoNombre
```



NO PODRÁ ACCEDER AL ATRIBUTO PRIVADO

```python
class SuperClase:
    # atributos y métodos de la clase base PROTEGIDOS
    def __init__(self, nombre):
        self._nombre = nombre

class SubClase(SuperClase):
    # atributos y métodos de la subclase
    def __init__(self, nombre):
        super().__init__(nombre)

    def modificarNombre(self, nuevoNombre):
        self._nombre = nuevoNombre
```

Herencia en python - sobreescritura de métodos

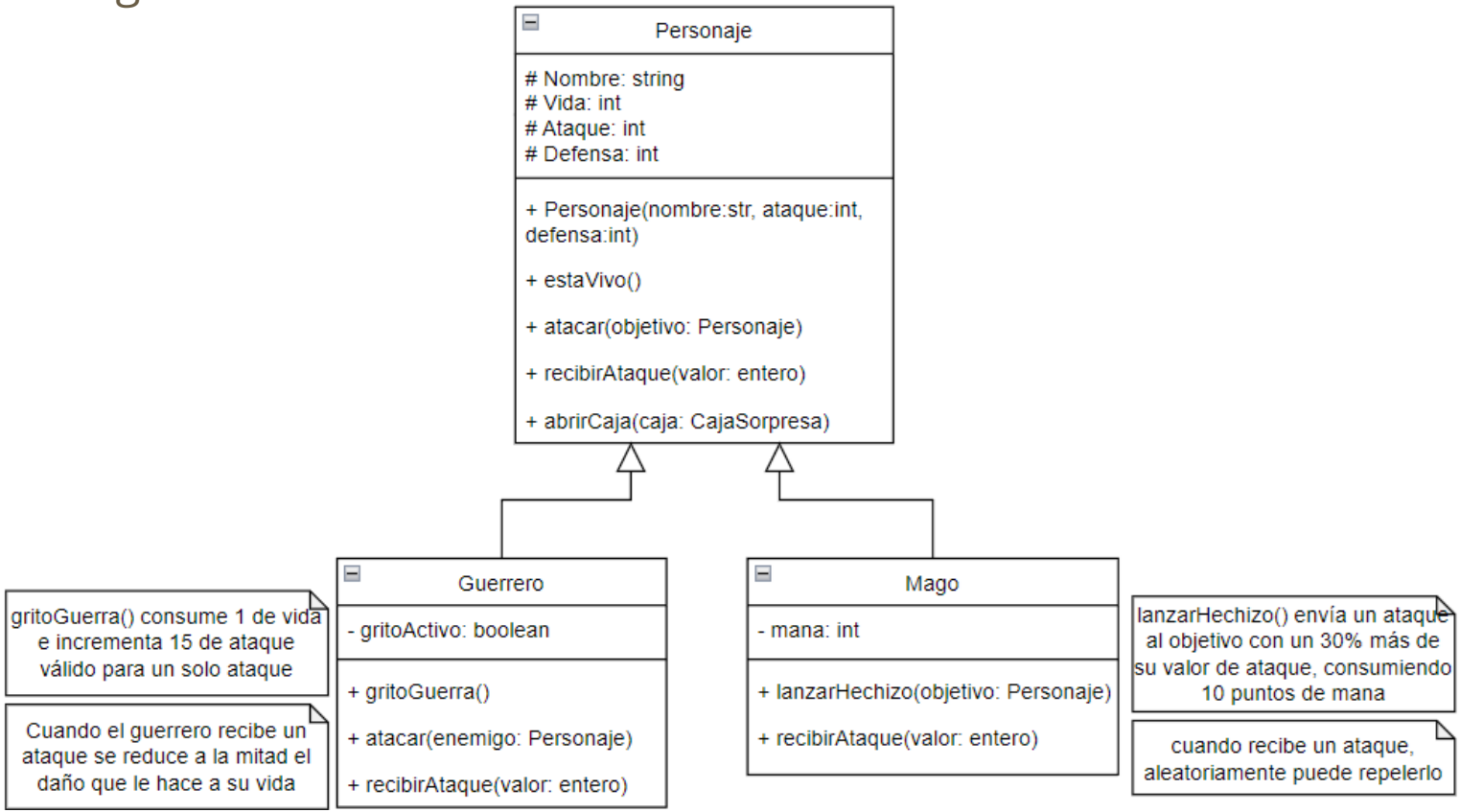
La sobreescritura permite redefinir un método existente en la clase base con un comportamiento nuevo para la subclase. Se utiliza cuando una subclase necesita modificar el comportamiento de un método de la clase base. Por ejemplo, el método `recibirAtaque()` se puede sobreescribir en la clase Guerrero para modificar la lógica de recibir daño.

```python
class SuperClase:
    # atributos y métodos de la clase base
    def __init__(self, nombre):
        self._nombre = nombre

    def obtenerNombre(self):
        return self._nombre
```

```python
class SubClase(SuperClase):
    # atributos y métodos de la subclase
    def __init__(self, nombre):
        super().__init__(nombre)

    def obtenerNombre(self):
        resultado = self._nombre + " (subclase)"
        return resultado
```

Caso de estudio: juego de rol simple

Implementemos el siguiente diagrama:



Clase Personaje

```python
from Arma import Arma   # Importamos la clase Arma desde el archivo Arma.py
from CajaSorpresa import CajaSorpresa, Caracteristica   # Importamos la clase CajaSorpresa desde el archivo CajaSorpresa.py

class Personaje:
    # Atributos de clase
    MAX_VIDA = 100
    MAX_ATAQUE = 50
    MAX_DEFENSA = 45
    MIN_VIDA = 0
    MIN_ATAQUE = 5
    MIN_DEFENSA = 0

    def __init__(self, nombre:str, ataque:int, defensa:int):
        """ Docstring . . . """
        if not isinstance(nombre, str) or nombre == "" or nombre.isspace():
            raise ValueError("El nombre debe ser un string válido.")
        if not isinstance(ataque, int) or ataque < Personaje.MIN_ATAQUE or ataque > Personaje.MAX_ATAQUE:
            raise ValueError(f"El ataque debe ser un número entero entre {Personaje.MIN_ATAQUE} y {Personaje.MAX_ATAQUE}.")
        if not isinstance(defensa, int) or defensa < Personaje.MIN_DEFENSA or defensa > Personaje.MAX_DEFENSA:
            raise ValueError(f"La defensa debe ser un número entero entre {Personaje.MIN_DEFENSA} y {Personaje.MAX_DEFENSA}.")
        self._nombre=nombre
        self._vida = Personaje.MAX_VIDA
        self._ataque=ataque
        self._defensa=defensa
        self._arma=None
```

Atributos protegidos

Herencia en python - Guerrero(Personaje)

```python
class Guerrero(Personaje):
    DANIO_GRITO = 15

    def __init__(self, nombre:str, ataque:int, defensa:int):
        """Inicializa los atributos de un guerrero."""
        super().__init__(nombre, ataque, defensa)
        self.__gritoActivo = False

    def gritoGuerra(self):
        """Activa el grito de guerra del guerrero.
        Consume 1 punto de vida, pero le permite incrementar el ataque 15 puntos en el próximo ataque."""
        if self._vida  > 1:
            self._vida -= 1
            self.__gritoActivo = True
```

Herencia en python - Guerrero(Personaje)

```
def atacar(self, enemigo:Personaje):
    """
    Ataca al enemigo especificado.
    Si no recibe un personaje lanza un ValueError.
    """
    if not isinstance(enemigo, Personaje):
        raise ValueError("El enemigo no puede ser None.")
    if self.estaVivo():
        if self.__gritoActivo:
            if self._arma != None:
                enemigo.recibirAtaque(self._ataque + self._arma.obtenerDanio()+ Guerrero.DANIO_GRITO)
            else:
                enemigo.recibirAtaque(self._ataque + Guerrero.DANIO_GRITO)
            self.__gritoActivo = False
        else:
            if self._arma != None:
                enemigo.recibirAtaque(self._ataque + self._arma.obtenerDanio())
            else:
                enemigo.recibirAtaque(self._ataque)
```

Herencia en python - Guerrero(Personaje)

```
def recibirAtaque(self, valorAtaque: int):
    """Recibe un ataque del enemigo."""
    if isinstance(valorAtaque, int):
        if self.estaVivo():
            if self._defensa < valorAtaque:
                # Si la defensa del personaje es menor al ataque recibido,
                # reduce a la mitad el valor del daño recibido.
                if self._vida - int((valorAtaque-self._defensa)/2) <= 0:
                    self._vida = Personaje.MIN_VIDA
                else:
                    self._vida -= int((valorAtaque-self._defensa)/2)
    else:
        raise ValueError("El valor de ataque debe ser un número entero positivo.")
```

Herencia en python - Mago(Personaje)

```python
from personaje import Personaje
import random

class Mago(Personaje):
    MAX_MANA=100
    PORCENTAJE_AUMENTO_ATAQUE=0.3

    def __init__(self, nombre:str, ataque:int, defensa:int):
        """Inicializa los atributos de un mago."""
        super().__init__(nombre, ataque, defensa)
        self.__mana =  Mago.MAX_MANA

    def obtenerMana(self)->int:
        """Devuelve el mana del mago."""
        return self.__mana
```

Herencia en python - Mago(Personaje)

```python
def lanzarHechizo(self, enemigo:Personaje):
    """Lanza un hechizo al enemigo especificado. Consume 10 puntos de mana.
    Si no recibe un personaje lanza un ValueError.
    """
    if not isinstance(enemigo, Personaje):
        raise ValueError("El enemigo no puede ser None.")
    if self.estaVivo():
        if self.__mana >= 10:
            enemigo.recibirAtaque(int(self._ataque * (1 + Mago.PORCENTAJE_AUMENTO_ATAQUE)))
            self.__mana -= 10
        else:
            raise ValueError("El mago no tiene suficiente mana para lanzar un hechizo.")

def recibirAtaque(self, valorAtaque: int):
    """Recibe un ataque del enemigo.
    Aleatoriamente el mago puede repeler el ataque y no recibir daño.
    Si no puede repelerlo recibe el daño normalmente.
    """
    # si el numero aleatorio es par, recibe el ataque
    if random.randint(1,10) % 2 == 0:
        super().recibirAtaque(valorAtaque)
    else:
        print(f"{self._nombre} ha repelido el ataque.")
```

Explicación del código anterior

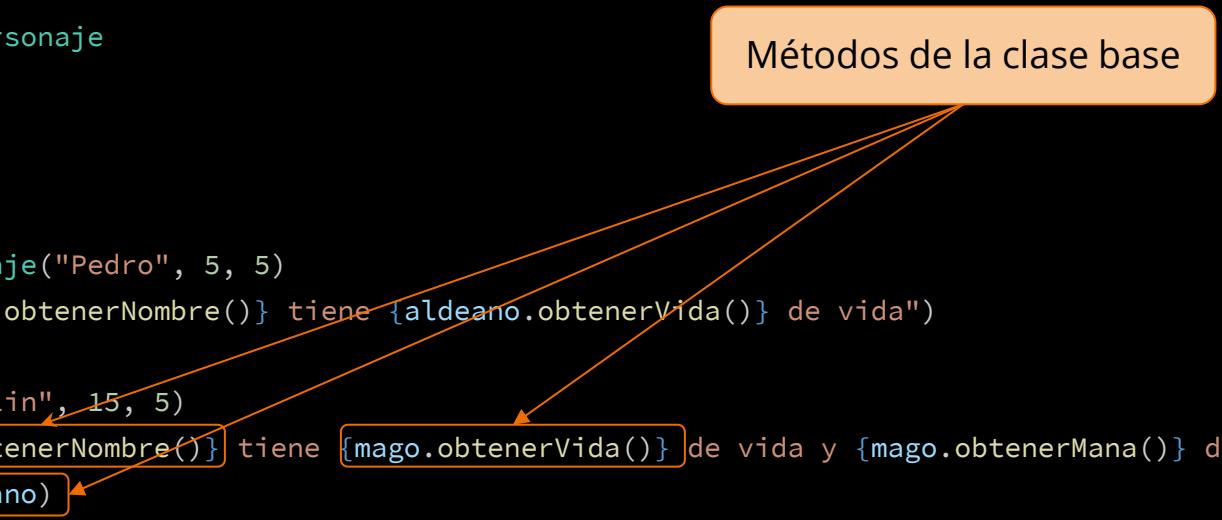
- Clase Base `Personaje`:
  - Define el comportamiento común de todos los personajes, como atacar, recibir daño, etc.
  - El constructor (`__init__`) inicializa los atributos nombre, vida, ataque, y defensa.
- Clase Derivada `Guerrero`:
  - Usa el constructor de la clase base mediante `super().__init__()` para evitar duplicar código, y extiende Personaje agregando un atributo que almacena el estado del grito de guerra (activo/inactivo).
  - Sobreescribe el comportamiento del método `recibirAtaque()`, que reduce el daño a la mitad, y el método `atacar()` que adapta el ataque si el grito de guerra está activo.
- Clase Derivada `Mago`:
  - Usa el constructor de la clase base mediante `super().__init__()` para evitar duplicar código, y extiende Personaje añadiendo un atributo extra '`mana`'.
  - Sobrescribe el comportamiento de `recibirAtaque()` permitiendo repelerlo.
  - Añade comportamiento al personaje mediante el método `lanzarHechizo()`.

Conceptos del código anterior

- Uso de `super()`: Permite invocar el constructor o métodos de la clase base desde una subclase, evitando la duplicación de código.
- Sobrescritura de Métodos: Las subclases pueden sobrescribir métodos de la clase base para ajustar comportamientos.
- Extensión de Atributos: Se pueden agregar atributos únicos a las subclases (como “`mana`” en `Mago`, o “`grito_activo`” en `guerrero`).
- Polimorfismo: las clases Mago y Guerrero se comportan distinto cuando reciben el mensaje `recibirAtaque()`. Esto es posible porque ambas clases (Guerrero y Mago) heredan de la misma clase base (Personaje), pero cada una puede sobrescribir o modificar el comportamiento de los métodos que heredan, manteniendo la interfaz común.

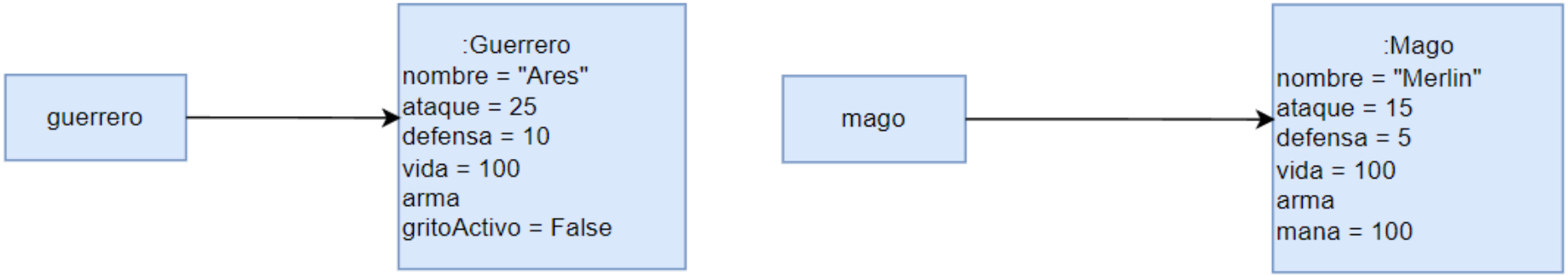
Herencia

```python
from mago import Mago
from personaje import Personaje

class Tester:
    @staticmethod
    def test():
        aldeano = Personaje("Pedro", 5, 5)
        print(f"{aldeano.obtenerNombre()} tiene {aldeano.obtenerVida()} de vida")

        mago = Mago("Merlin", 15, 5)
        print(f"{mago.obtenerNombre()} tiene {mago.obtenerVida()} de vida y {mago.obtenerMana()} de mana")
        mago.atacar(aldeano)
```



Un objeto de clase especializada puede recibir mensajes considerando los métodos que brindan su propia clase y la clase base.

Herencia



El estado interno de un objeto de clase Guerrero tiene todos los atributos de Personaje más los específicos de su clase.

Ídem para el estado interno de la clase Mago que especializa a Personaje.

Caso de estudio: juego de rol simple