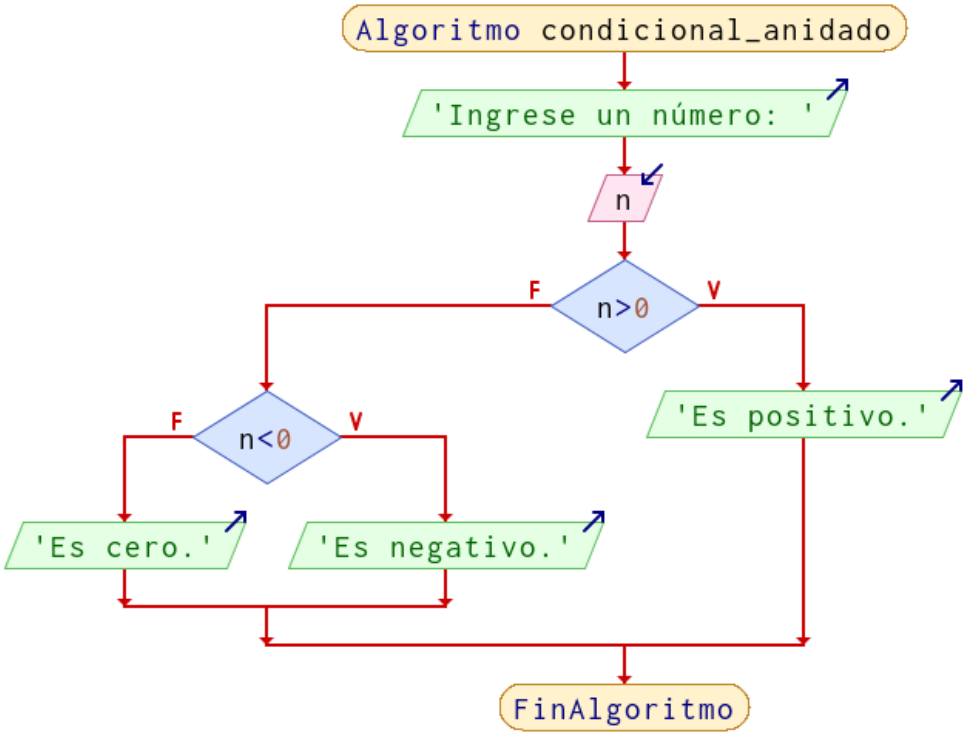
1. El modelo procedural vs. orientado a objetos

En el modelo del paradigma orientado a procesos se planifican procesos y secuencias de pasos, bifurcaciones y bucles que muchas veces se plantean de formas extremadamente específicas, aún utilizando la metodología *top-down*. Esto supone de un gran esfuerzo para lograr sistemas de gran tamaño, que a la vez, tengan alta cohesión y bajo acoplamiento necesarios para que sean escalables[1].

Figura 1: Ejemplo de diagrama de flujo de un software modelado de forma procedural.



En lugar de plantear un consumo de estructuras, manipulando y operando sobre sus datos, el paradigma de objetos plantea un modelado de la realidad donde cada uno de estos tenga propiedades y comportamientos característicos y relacionándose e interactuando entre sí *enviándose mensajes*. De esta manera, los mismo objetos saben como comportarse con respecto a sus *atributos*. A este **principio** se lo conoce como *tell don't ask* y es fundamental en este paradigma.

En la gran mayoría de los lenguajes Orientados a Objetos (OO en adelante), existen dos conceptos básicos:

[1] La escalabilidad de un sistema de software hace referencia a que tan sencillo es agregar nuevos módulos o soportar nuevas características sin la necesidad de modificar lo codificado previamente.

Objeto: Es una unidad que tiene un **estado**, **comportamiento** e **identidad**. Un objeto caracteriza tanto entidades físicas (auto, mesa) como abstractas (fecha, virtud). El estado hace referencia al valor de sus **atributos**, su comportamiento a sus **métodos** (funciones) y su identidad a los nombres de los mismos.

Clase: Es un modelo o molde que describe como se caracterizan los objetos para poder ser creados (instanciados). En estas se declaran los atributos y métodos que son comunes a todos los objetos de la clase en cuestión.

En el ejemplo de código expuesto en el Programa 1, vemos como se declara una *clase* `Dado` (de 6 caras) que modela solamente el comportamiento del mismo a través del *método* `tirar()`. Luego en el cuerpo principal del *script* se *instancia* el objeto `d` y luego se envía el mensaje `tirar()` que es impreso por pantalla a través de la función estándar `print()`.

Programa 1: Declarar e instanciar un objeto en `Python`

```
1  import random # módulo que contiene la función randint()
2
3  class Dado:
4    """ Modela un dado de 6 caras """
5
6    def tirar(self) -> int:
7      """ Retorna un entero al azar del intervalo [1, 6] """
8      return random.randint(1,6)
9
10 if __name__ == "__main__":
11   d = Dado()         # Se crea el objeto 'd' de la clase 'Dado'
12   print(d.tirar())   # se tira el 'Dado' 'd'
```

En `Python` las clases se declaran utilizando la palabra reservada `class` seguida del nombre de la misma, empezada en mayúscula por convención y utilizando *UpperCamelCase*. Luego, los métodos son simples declaraciones de funciones (utilizando la palabra reservada `def`) con la particularidad de que deben tener como parámetro obligatorio una variable llamada comúnmente `self` que hace referencia al mismo objeto[2].

Luego, en la parte principal del *script* (a partir de la *línea 8*), se instancia el objeto único `d` y se envía el mensaje `tirar()` como se indicó anteriormente.

[2]Además, en este curso se alentará la utilización de *type hints*, es decir especificar el tipo de dato esperado de los parámetros y el que retornan la funciones/métodos. Esto se indica con dos puntos `:` para los parámetros y con un guion seguido por un corchete angular `->` para los tipos de datos retornados.

En este modelo los dados solo poseen comportamiento (un método `tirar()`), no tienen ninguna propiedad o estado. Si se quisiera flexibilizar el software y de pronto poder crear dados de diferentes cantidades de caras, entonces se podría plantear de la siguiente manera.

Programa 2: Declarar e instanciar un objeto con atributos en `Python`

```python
import random # módulo que contiene la función randint()

class Dado:
    """ Modela un dado de N caras """
    def __init__(self, n_caras : int) -> None:
        """Construye un Dado de N caras

        Args:
            n_caras (int): Cantidad de caras del dado.
        """
        # inicialización del atributo '__n_caras'
        self.__n_caras = n_caras

    def tirar(self) -> int:
        """ Retorna un entero al azar del intervalo [1, N] """
        return random.randint(1, self.__n_caras)

if __name__ == "__main__":
    d1 = Dado(6)       # Se crea un dado de 6 caras
    print(d1.tirar())
    d2 = Dado(24)      # Se crea un dado de 24 caras
    print(d2.tirar())
```

Como se observa en el Programa 2, se ha agregado un nuevo método, pero no es un método cualquiera, es un *método especial*[3]. Este en particular se llama **inicializador** de la clase. Allí se **agregarán e inicializarán los atributos** de la instancia. En este caso, se recibe por parámetro una variable `n_caras` del tipo entero y se asigna al atributo con el mismo nombre, pero con la particularidad de que este empieza con doble guion bajo '`__n_caras`'. Esto se explicará más adelante en la unidad de *encapsulamiento*.

Los **atributos** en `Python` se declaran habitualmente dentro del inicializador anteponiendo la variable `self.` (punto) y luego el nombre, que como se mencionó anteriormente se escribirá con un doble guion bajo por delante.

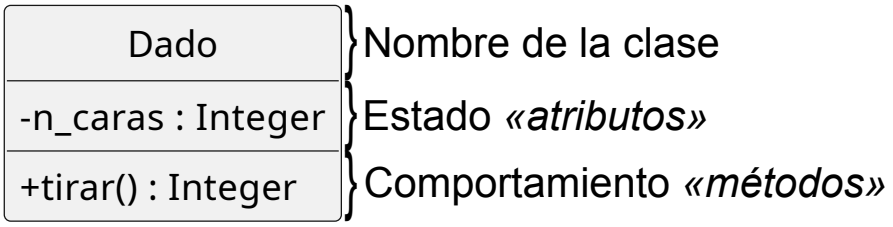
[3]En `Python` existe varios de estos **métodos especiales** que tienen nombres bien definidos y todos se encuentran encerrados entre doble guion bajo. A medida que sean de utilidad se irán mencionando, por ahora, solo se utilizarán los inicializadores `__init__(self[, ...])`.

2. Lenguaje Unificado de Modelado (UML)

Para diseñar software de forma ordenada y escalar es necesario hacer un planteo del sistema. En el paradigma OO, se analiza el problema y se hace una abstracción del mismo al igual que en el procedural. La diferencia más importante radica en que en lugar de plantear una secuencia de procesos, se modelan objetos que interactúan entre sí.

Con tal fin, y de manera en que los desarrolladores puedan comunicarse entre sí, más allá del lenguaje de programación que utilicen, existe un estándar de modelado llamado UML[4] que abarca diferentes aspectos del desarrollo en sus diversos diagramas. El que nos ayuda a entender el planteo del sistema es el **diagrama de clases**. En este tipo de gráfico las clases se modelan de la siguiente forma.

Figura 2: Representación de una clase en UML.



Se observa en la Figura 2 que los objetos se modelan en cajas con 3 partes. La primera es el nombre, la segundo los atributos y la última los métodos.

En el diagrama de clases no debe hacerse referencia a ningún lenguaje en particular, ya que es un *gráfico universal*. Si bien podríamos agregar el inicializador __init__ (que es un nombre muy particular de Python), en este caso no aportaría demasiada información, ya que lo único que hace es inicializar el único atributo existente. Por este mismo motivo, tampoco se incluirá el parámetro `self` en el diagrama.

Como se mencionó anteriormente, existen varios tipos de diagramas en UML, pero a lo largo de este curso se utilizarán **diagramas de clases** –como el anterior– y **modelado de Casos-Uso**, aunque eventualmente pueda hacerse uso de algún otro para clarificar situaciones como podrían ser los *diagramas de actividad* y *diagramas de secuencia*.

[4]Para más información consulte https://uml.org

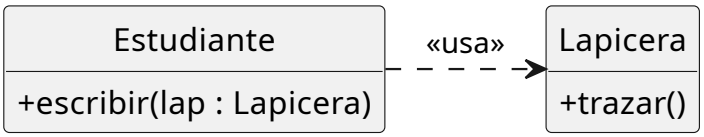
2.1. Relaciones entre objetos

Como se ha mencionada anteriormente, uno de los puntos importantes del paradigma OO es el planteo de un diseño donde objetos interactúan entre sí y están relacionados. En `UML` se pueden plantear estas relaciones en forma de flechas que conectan las clases entre sí. Dependiendo de la forma que estas tengan comunican diferentes relaciones. Existen 6 tipos: *dependencia, asociación, agregación, composición, generalización y realización.*

2.1.1. Dependencia

Esta es la relación más débil, significa que una clase usa otra clase o elemento de la misma. Esto se ve reflejado en el código cuando una clase recibe por parámetro o utiliza internamente en alguno de sus métodos a otra clase o elemento dentro de la misma. Se simboliza con una flecha discontinua con una punta abierta.

Figura 3: Dependencia



Esto en código `Python` se vería como se observa en Programa 3, donde los objetos de la clase `Estudiante` *usan* un objeto de la clase `Lapicera` a través del método `escribir()`.

Programa 3: Dependencia

```python
class Lapicera:
  def trazar(self):
    ...

class Estudiante:
  def escribir(self, lap : Lapicera):
    ...
```

Nota

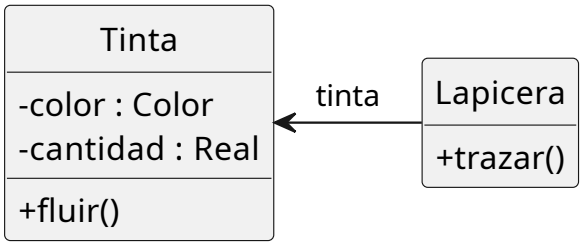
Es muy común que en los diseños haya muchas dependencias, por esta razón, suelen obviarse de los diagramas para clarificarlos. De lo contrario, estos tendrían excesivas líneas cruzándose entre sí. Sin embargo, **deben utilizarse** si la relación es relevante para lo que se desea comunicar.

2.1.2. Asociación

En este tipo de relación una clase tiene un atributo que es instancia de otra clase. El nombre del mismo suele dibujarse sobre la línea llena con punta abierta, y en caso de ser un arreglo o conjunto se agrega además la cardinalidad.

Esta relación significa que la clase "tiene" (*has-a*), "viaje en", "visita a", etc.

Figura 4: Asociación



Es importante destacar que cuando el atributo está representado por la relación (la flecha), esta no debe agregarse en el estado, ya que sería un duplicado en la notación. En cambio, si la clase no forma parte del diagrama actual, por ejemplo, la clase `Color`, entonces sí es necesaria su aparición en el estado de la clase `Tinta`.

En `Python` esto se vería como se observa en el Programa 4. En cuanto a la relación de `Color` con respecto a `Tinta`, podríamos entender que también es una asociación.

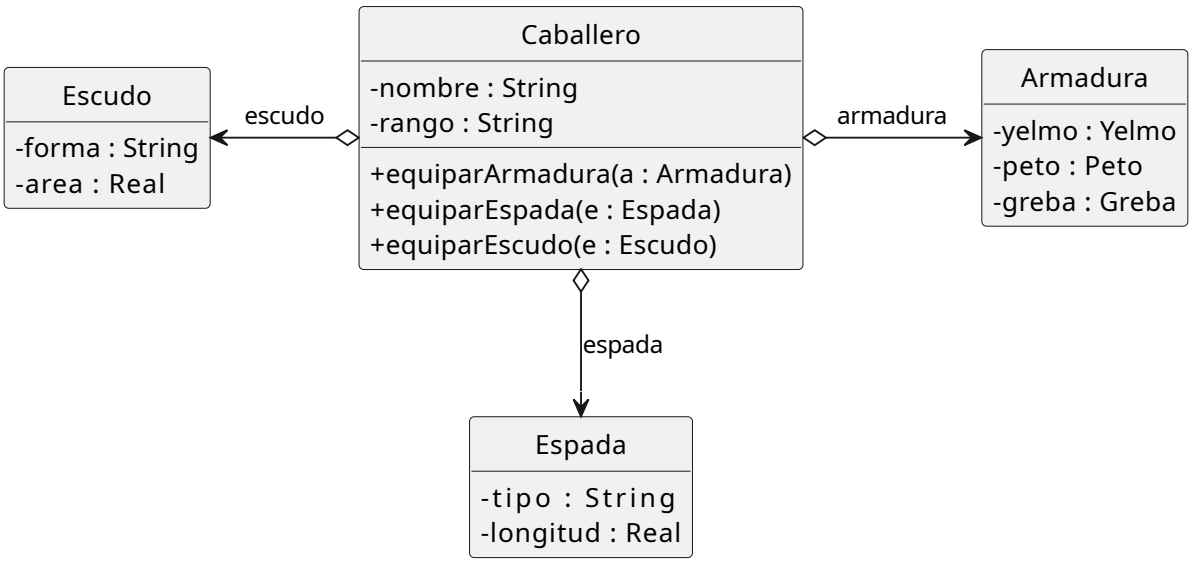
Programa 4: Asociación

```
class Tinta:
  def __init__(self) -> None:
    self.__color = Color()
    self.cantidad = 1.15 # Nro Real => float

  def fluir(self):
    ...

class Lapicera:
  def __init__(self) -> None:
    self.__tinta = Tinta() # Asociación

  def trazar(self):
    ...
```

2.1.3. Agregación

En esta relación podemos decir que una clase está compuesta de atributos que son instancias independientes de otras clases. Esto quiere decir que los objetos "que componen" existen por fuera de la clase y son agregados al construir el objeto o bien pasados por parámetro de algún mensaje. Esta relación se traza con una linea continua, punta abierta y un rombo vacío desde la clase que contiene el atributo hacia la que compone la agregación.

Figura 5: Agregación



En este ejemplo, se observa como un Caballero tiene nombre, rango y está compuesto por una armadura, una espada y un escudo, los cuales pueden ser agregados a través de los métodos correspondientes. En `Python` esto se vería como en el Programa 5.

Programa 5: Agregación

```
class Armadura:
  def __init__(self) -> None:
    self.__yelmo : Yelmo
    self.__peto  : Peto
    self.__greba : Greba

class Espada:
  def __init__(self, tipo : str, longitud : float) -> None:
    ...

class Escudo:
  def __init__(self, forma : str, area : float) -> None:
    ...
```

```
class Caballero:

  def __init__(self, nombre : str, rango : str) -> None:
    self.__nombre   = nombre
    self.__rango    = rango
    self.__armadura : Armadura
    self.__espada   : Espada
    self.__escudo   : Escudo

  def equipar_armadura(self, a : Armadura):
    self.__armadura = a

  def equipar_espada(self, e : Espada):
    self.__espada = e

  def equipar_escudo(self, e : Escudo):
    self.__escudo = e

if __name__ == "__main__":
  art = Caballero("Arturo", "Rey")
  excalibur = Espada("Mítica", 1.2)
  art.equipar_espada(excalibur)
  ...
  lan = Caballero("Lancelot", "Paladín")
  esp_normal = Espada("Normal", 1.05)
  lan.equipar_espada(esp_normal)
  ...
```

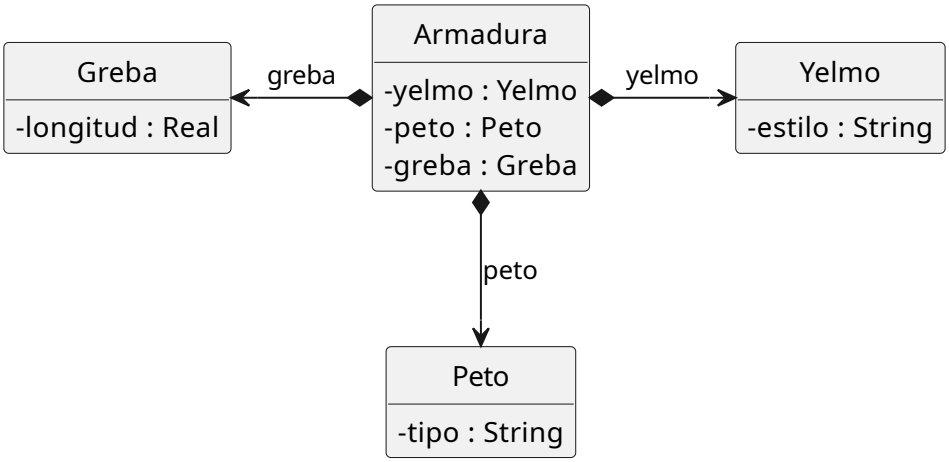
Como se observa, la armadura, la espada y el escudo son objetos independientes, que existen por sí mismos y son *agregados* a instancias de `Caballero` a través del método correspondiente. Como por ejemplo, en el caso de los objetos `excalibur` con `art` y `esp_nomral` con `lan`. Inicialmente, los atributos no tiene un valor, solo se indica del tipo de dato que es en el inicializador de `Caballero`.

2.1.4. Composición

Este tipo de relación es parecida a la agregación, aquí la clase también está compuesta de atributos, pero en este caso las instancias son creadas dentro de la compuesta. Es decir que el ciclo de vida del atributo es compartido por el del objeto en sí. Este tipo de relación se traza con una línea continua, punta abierta y un rombo lleno desde la clase que contiene el atributo hacia la que compone al objeto.

En este caso, una `Armadura` está compuesta por instqancias de un `Yelmo`, un `Peto`

Figura 6: Composición



y una `Greba`. Estos existen mientras exista la armadura y dejarán de existir cuando esta desaparezca. Es decir que son creados en el inicializador de la clase, esto podemos implementarlo en `Python` como se observa en el Programa 6.

Programa 6: Composición

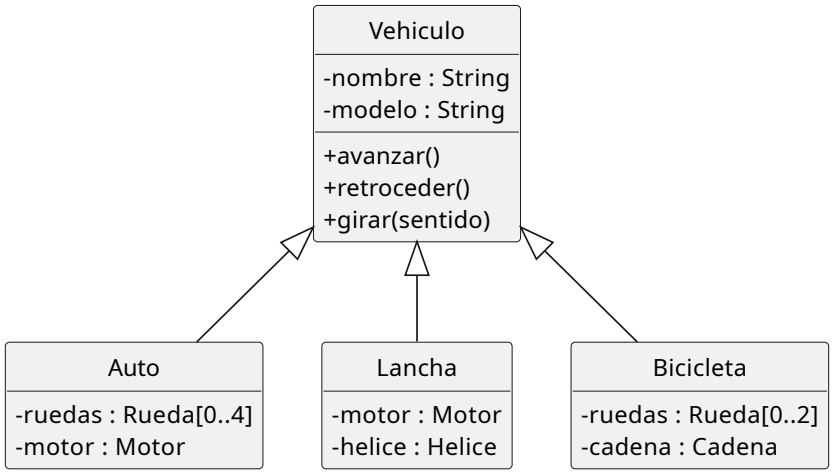
```
class Yelmo:
  def __init__(self, estilo : str) -> None:
    self.__str = estilo

class Peto:
  def __init__(self, tipo : str) -> None:
    self.__tipo = tipo

class Greba:
  def __init__(self, longitud : float) -> None:
    self.__longitud = longitud

class Armadura:
  def __init__(self) -> None:
    self.__yelmo = Yelmo("Capellina")
    self.__peto  = Peto("Brigantina")
    self.__greba = Greba(0.45)
```

En este ejemplo, se puede crear un solo tipo de armadura, la cual tiene un *yelmo capellina*, un *peto brigantino* y una *greba de 45cm*, los cuales son instanciados en el método `__init__`.

2.1.5. Generalización

Una clase derivada es un (*is-a*) tipo especial de otra clase más general. Los objetos de la clase derivada "heredan" los miembros no privados de la clase base o *superclase* y pueden ser usados en contextos que requieran objetos de esta última. Ésta relación es representada por una flecha de línea continua con punta en triangulo, la cual apunta a la clase más genérica o *superclase*.

Figura 7: Generalización



En este ejemplo podemos observar como un `Auto` *es un* `Vehículo`, una `Lancha` *es un* `Vehículo` y una `Bicicleta` *es un* `Vehículo`.

Esto en `Python` se logra poniendo el nombre de la clase general o *superclase* entre paréntesis de la clase derivada o *subclase* como se observa en el Programa 7.

Programa 7: Generalización

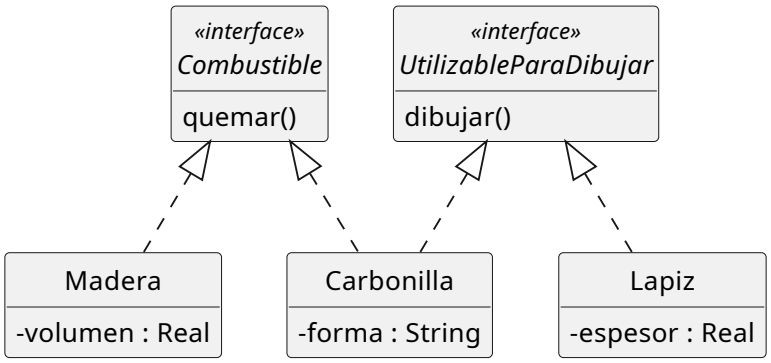
```
class Vehiculo:
  ...

class Auto(Vehiculo):
  ...

class Lancha(Vehiculo):
  ...

class Bicicleta(Vehiculo):
  ...
```

Se verán más detalles sobre tema cuando se desarrolle en el apunte de **Herencia**.

2.1.6. Realización

Este tipo de relación tiene cierto parecido con la Generalización, pero en este caso las clases se *comprometen* a implementar un contrato conocido como **interfaz**. Esta relación está representada por una flecha de línea discontinua con punta en triangulo.

Figura 8: Realización



En este ejemplo la `Madera` y la `Carbonilla` se comprometen a implementar el mensaje `quemar()`, por otro lado `Lapiza` y `Carbonilla` se comprometen a implementar `dibujar()`. Note que la `Carbonilla` se compromete a cumplir con dos contratos distintos.

En `Python` no existen formalmente las interfaces como en otros lenguajes (p.e. Java, C#, Go, etc.). Pero posee un módulo que *emula* dicha funcionalidad llamado `ABC` (*Abstract Base Classes*). Para poder utilizarlo debe importarse y luego se colocar entre paréntesis el nombre de la clase que implementa `ABC` (como en la herencia).

Programa 8: Generalización

```
from abc import ABC # Abstract Base Classes

class Combustible(ABC):
  @abstractmethod
  def quemar():
    ...

class UtilizableParaDibujar(ABC):
  @abstractmethod
  def dibujar():
    ...

class Madera(Combustible):
  ...
```

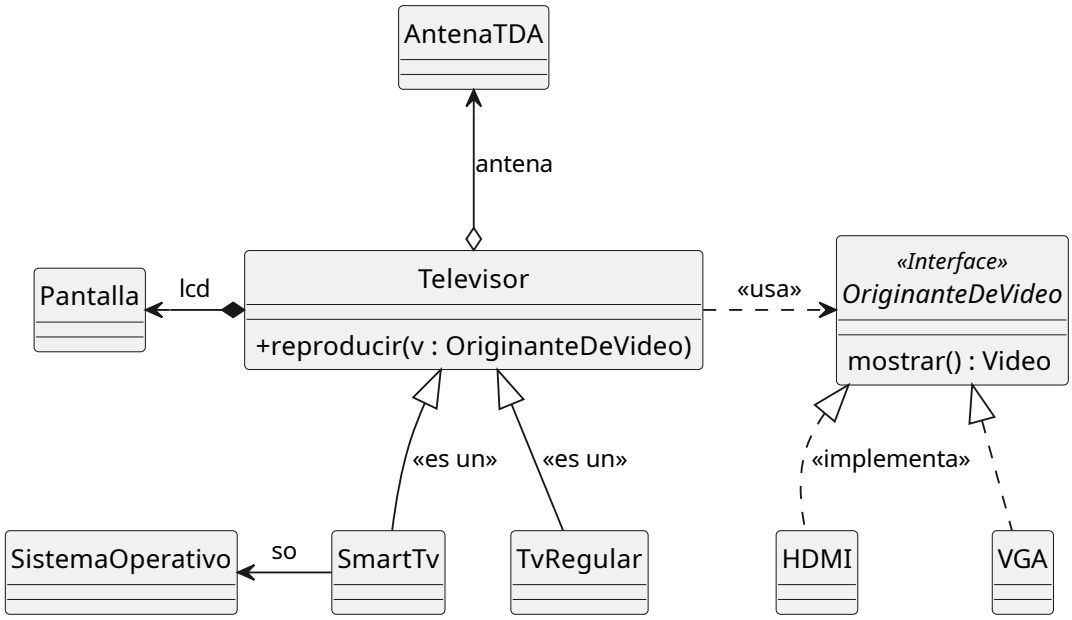
```
class Lapiz(UtilizableParaDibujar):
  ...

class Carbonilla(Combustible,UtilizableParaDibujar):
  ...
```

Los detalles de este mecanismo también serán estudiados en el apunte referente a **Herencia**.

2.2. El diseño OO

Un diseño de software OO está compuesto en parte por diagramas de clases que ponen en juego todas las relaciones aquí expuestas. Podemos observar un ejemplo en Figura 9.

Figura 9: Diseño de software OO



3. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: `https://docs.python.org/`
- UBA – Algoritmos I: `https://algoritmos1rw.ddns.net/`
- Interprete Visual de Python: `http://www.pythontutor.com/visualize.html`