Programación 2

Herencia y Polimorfismo
Clase Abstracta

Calidad de Software

La calidad de un producto de software puede definirse como su capacidad para satisfacer los requisitos establecidos durante el desarrollo de requerimientos.

La calidad puede medirse de acuerdo a distintos factores.

Algunos de estos factores son percibidos por el usuario o cliente. Otros factores son transparentes para el usuario o cliente, aunque por supuesto lo afectan indirectamente.

Calidad de Software

Correctitud

Un producto de software correcto actúa de acuerdo a los requerimientos especificados.

Eficiencia

Un producto de software es eficiente si tiene una baja demanda de recursos de hardware, en particular tiempo de CPU, espacio de memoria y ancho de banda.

Portabilidad

Un producto de software es portable si puede ejecutarse sobre diferentes plataformas de hardware y de software.

Calidad de Software

Simplicidad

Un producto de software es simple si es fácil de usar, su interfaz es amigable y no requiere demasiado entrenamiento ni capacitación por parte del usuario.

Robustez

Un producto de software es robusto se reacciona adecuadamente aun en circunstancias no especificadas en los requerimientos.

Legibilidad

La legibilidad está fuertemente ligada a modularización y la estructura del código. La legibilidad impacta en la reusabilidad y la extensibilidad.

Productividad de Software

Extensibilidad

Un producto de software es extensible si es fácil adaptarlo a cambios en la especificación de requerimientos. El primer requerimiento para la extensibilidad es la legibilidad.
Cuando hablamos de que es fácil de adaptar significa que pueden agregarse nuevos módulos con modificaciones mínimas sobre los anteriores.

Reusabilidad

Un módulo de software es reusable si puede utilizarse para la construcción de diferentes aplicaciones.
Por ejemplo un módulo que brinde servicios para dibujar figuras geométricas puede ser usado en diferentes aplicaciones.

La herencia y el polimorfismo favorecen la productividad.

Programación Orientada a Objetos

La POO mejora la calidad y la productividad a través de los conceptos de:

- **abstracción de datos y encapsulamiento**, favorecen fundamentalmente la correctitud, robustez, legibilidad y reusabilidad
- **herencia y polimorfismo**, favorecen fundamentalmente la reusabilidad y extensibilidad

Programación Orientada a Objetos

Un sistema desarrollado en base al paradigma de programación orientada a objetos está conformado principalmente por una colección de clases vinculadas por relaciones de:

- **Dependencia** una clase crea objetos o recibe objetos de otra clase como parámetros.
- **Asociación** una clase incluye atributos de clase o de instancia de otra clase.
- **Herencia jerárquica** una clase especializa a otra de modo que los objetos de la clase especializada son también objetos de otra clase más general.

Herencia

La **herencia jerárquica** es un mecanismo que permite organizar clases de acuerdo a relaciones de **generalización-especialización**.

Una clase puede ser usada para crear instancias en ejecución, pero también para definir otras clases más específicas y especializadas.

Cada clase que se usa para definir otra más específica se llama **clase base o superclase**.

Toda clase que hereda de una clase base se llama **clase derivada o subclase**.

Herencia

La **herencia jerárquica** se caracteriza porque cada clase puede **derivar** en varias subclases, en python puede tener varias clase padre (herencia múltiple), y puede tener varios ancestros a pesar de tener una única clase padre.

Las clases relacionadas por herencia pueden dibujarse como un **árbol**, cuya **raíz** es la clase más general.

Las **hojas** del árbol son las clases más especializadas.

Las clases de los niveles intermedio son al mismo tiempo clases base y derivadas.

Herencia y Abstracción

La abstracción de datos permite clasificar objetos en clases.

La herencia jerárquica aumenta el nivel de abstracción porque las clases son a su vez clasificadas a partir de un proceso de generalización o especialización.

Herencia y Encapsulamiento

El encapsulamiento permite ocultar los detalles de la definición de una clase, mostrando sólo aquellos elementos que permiten crear y manipular objetos.

La interfaz de una clase está constituida por todos los miembros que van a ser visibles desde otras clases.

Se utiliza el modificador de acceso “protegido” para establecer que las clases derivadas tienen acceso a los miembros protegidos de sus clases ancestro (en python se utiliza un guión bajo al inicio del nombre del atributo).

Los atributos que definamos como privados (doble guión bajo al inicio del nombre) no podrán ser accedidos desde las clases derivadas.

Herencia y Encapsulamiento

Existen diferentes **criterios** referidos al **nivel de encapsulamiento** que debería ligar a clases vinculadas por una relación de herencia.

Un argumento a favor de que las clases derivadas accedan a todos sus atributos, es que una instancia de una clase específica es también una instancia de las clases más generales de modo que debería poder acceder y modificar su estado interno.

El argumento en contra es que si se modifica la implementación de la clase base, el cambio afectará a todas las clases derivadas que accedan directamente a la representación.

Herencia y Encapsulamiento - Nivel de encapsulamiento

Atributos privados y acceso a través de métodos:

- Encapsulamiento: Los atributos se mantienen privados, lo que protege los datos internos de la clase y evita modificaciones accidentales.
- Control sobre las modificaciones: Los métodos de la clase base controlan cómo se acceden y modifican los atributos, lo que permite realizar validaciones y cálculos antes de permitir cambios.
- Flexibilidad: Si se necesita cambiar la forma en que se almacenan o calculan los datos, solo se modifican los métodos de la clase base, sin afectar a las clases derivadas.

Atributos protegidos:

- Acceso directo: Las subclases pueden acceder directamente a los atributos protegidos, lo que simplifica el código en algunos casos.
- Menor control: Se pierde cierto control sobre cómo se modifican los atributos, ya que las subclases pueden cambiarlos directamente sin pasar por los métodos de la clase base.
- Menor encapsulamiento: Los atributos protegidos son menos encapsulados que los privados, lo que puede hacer el código más susceptible a errores.

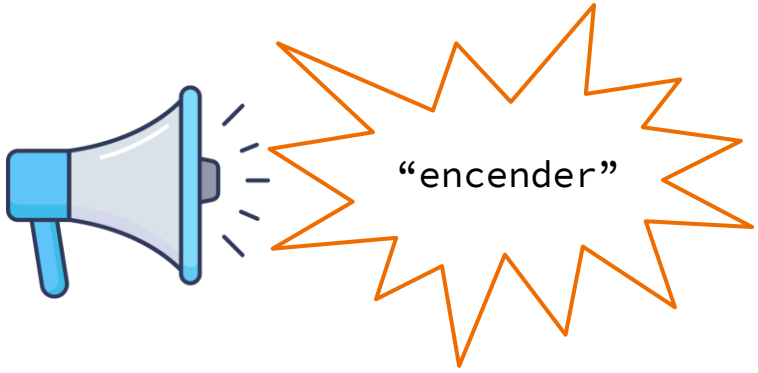
Polimorfismo

El concepto de polimorfismo es central en la programación orientada a objetos.

Polimorfismo significa muchas formas y en ciencias de la computación en particular se refiere a “la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál corresponde usar”.

En el contexto de la programación orientada a objetos el polimorfismo está relacionado con variables, asignaciones y métodos.

Polimorfismo




ATENCION
Las siguientes diapositivas no entran en el parcial, pero contienen conceptos importantes.

Polimorfismo en lenguajes fuertemente tipados

Una **variable polimórfica** puede quedar asociada a objetos de diferentes clases. Es decir, puede referirse a objetos de distintas clases, siempre y cuando esas clases tengan una relación jerárquica (por ejemplo, heredan de una clase base común). Esto es lo que permite que una misma variable pueda manejar diferentes tipos de objetos bajo una interfaz común.

El tipo de la declaración, llamado **tipo estático**, determina el conjunto de tipos de los objetos que pueden ligarse, esto es el conjunto de **tipos dinámicos**.

El tipo estático se conoce en compilación, el tipo dinámico se liga en ejecución.

```
// En Java
Animal miAnimal = new Perro();
```

Polimorfismo en lenguajes fuertemente tipados

Una **asignación polimórfica** liga un objeto de una clase a una variable declarada de otra clase.

Este es el mecanismo que permite que una variable de tipo de clase base se asocie con un objeto de una clase derivada. El polimorfismo por asignación se basa en la capacidad de que un objeto de una subclase pueda ser tratado como si fuera un objeto de la superclase. En lenguajes como Java o C#, esto es esencial para la reutilización de código y la implementación de interfaces genéricas.

```
// En Java
Vehiculo miVehiculo = new Bicicleta();  // La variable miVehiculo apunta a un objeto de la clase Bicicleta.
Auto otroVehiculo = new Auto();         // otroVehiculo es un objeto de la clase Auto.
miVehiculo = otroVehiculo;              // Ahora miVehiculo apunta a un objeto de la clase Auto.
```

Polimorfismo en lenguajes fuertemente tipados

Un **método polimórfico** incluye una o más variables polimórficas como parámetro.

Un método polimórfico acepta como parámetros variables que pueden referirse a objetos de distintas clases (generalmente clases que heredan de una clase común). En lenguajes fuertemente tipados, los métodos polimórficos son fundamentales para el uso de técnicas como la sobrescritura de métodos (overriding) y la herencia, permitiendo que las clases derivadas proporcionen su propia implementación de un método definido en la clase base.

```
// en Java - Vehiculo puede ser Auto o Bicicleta
public void moverVehiculo(Vehiculo v) {
        v.moverse();
}
```

Polimorfismo en lenguajes fuertemente tipados

En lenguajes como Java o C#, es crucial que las clases compartan una relación de herencia para que el polimorfismo sea posible.

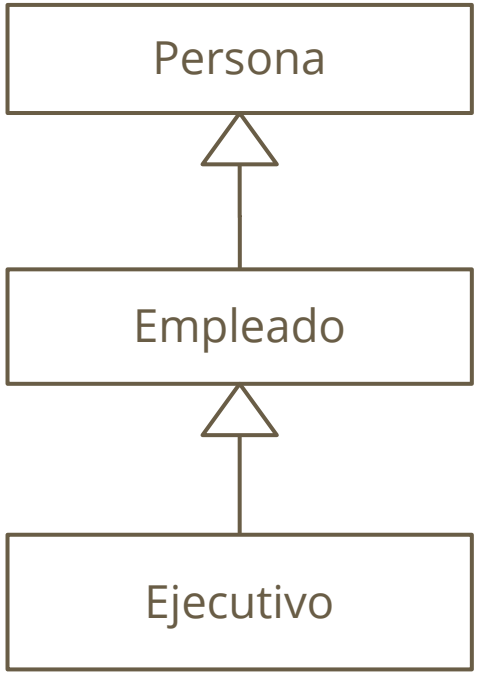
Además, en los lenguajes fuertemente tipados la variable polimórfica está restringida a referirse a objetos cuyo tipo sea una subclase (o la propia clase) de su tipo declarado, lo que proporciona seguridad en el manejo de tipos en tiempo de compilación.

El **tipo estático** de una variable: es el tipo que aparece en la declaración.

El **tipo dinámico** de una variable: se determina en ejecución y corresponde a la clase a la que pertenece el objeto referenciado.

El tipo estático de una variable determina el conjunto de tipos dinámicos a los que puede quedar asociada y los mensajes que puede recibir.

Polimorfismo en lenguajes fuertemente tipados



Tipo Estático	Tipo Dinámico
Persona	Persona, Empleado, Ejecutivo
Empleado	Empleado, Ejecutivo
Ejecutivo	Ejecutivo

En un lenguaje fuertemente tipado, una vez que se declara una variable con un tipo específico (tipo estático), esa variable solo puede almacenar objetos de ese tipo o de sus subclases en ejecución (tipo dinámico).


FIN DE LAS DIAPOSITIVAS QUE
NO ENTRAN EN EL PARCIAL

Clase abstracta

En el diseño de una aplicación es posible definir una clase que factoriza propiedades de otras clases más específicas, sin que existan en el problema objetos concretos vinculados a esta clase más general.

En este caso la clase se dice abstracta porque fue creada artificialmente para lograr un diseño que modele la realidad.

En ejecución no va a haber objetos de software de una clase abstracta.

Una clase abstracta es una clase de la cual no se pueden definir instancias (crear objetos).

Clase abstracta

Una clase abstracta puede incluir uno, varios, todos o ningún método abstracto.

Si una clase hereda de una clase abstracta y no implementa todos los métodos abstractos, también debe ser definida como abstracta.

Si una clase posee un método abstracto, la clase se considera abstracta.

Una clase concreta debe implementar todos los métodos abstractos de sus clases ancestro. De lo contrario debe definirse como abstracta.

El constructor de una clase abstracta sólo va a ser invocado desde los constructores de las clases derivadas.

La notación UML para clases abstractas consiste en escribir en letra cursiva el nombre de dicha clase

Clase abstracta en python

Python implementa clases abstractas a través del **módulo abc** (*Abstract Base Classes*). Estas clases contienen uno o más métodos abstractos, que son métodos que deben ser implementados en las subclases. **Si una subclase no implementa todos los métodos abstractos, no puede ser instanciada**.

Características de las clases abstractas en Python:

- No pueden ser instanciadas directamente: Solo se puede crear una instancia de una subclase que haya implementado todos los métodos abstractos.
- Definen una interfaz común: Las clases abstractas proporcionan una interfaz que todas las subclases deben seguir.
- Métodos abstractos: Son métodos declarados en la clase abstracta, pero sin implementación (no tienen cuerpo). Las subclases deben implementar estos métodos.

Clase abstracta en python

Implementación de clases abstractas

Para crear una clase abstracta en Python, se procede de la siguiente manera:

- Importar el **módulo abc**.
- Heredar de **abc.ABC** para indicar que es una clase abstracta.
- Decorar los métodos abstractos con el decorador **@abstractmethod**.

Lo veremos con un ejemplo a continuación.

Caso de estudio: Agencia Publicitaria

Una agencia publicitaria publica avisos en diferentes medios de comunicación: televisión, radio, diarios y revistas.

Cada aviso tiene asociado un nombre de fantasía, un producto, una empresa, la fecha de inicio y una duración en días.

Las campañas en radios y televisión tienen una emisora, una duración en segundos y una cantidad de repeticiones por día. No hay dos avisos de radio o TV con el mismo nombre, de una misma empresa. Los atributos nombre-empresa son la **clave** .

Caso de estudio: Agencia Publicitaria

Los avisos publicados en diarios y revistas tienen un título, una cantidad de centímetros cuadrados de texto. No hay dos avisos impresos con el mismo nombre, de una misma empresa. Los atributos nombre-empresa son la **clave** .

El costo de una campaña en radio o televisión se calcula como el producto entre la cantidad de días que dura la campaña, por la cantidad de repeticiones por día, por la duración en segundos, por un monto fijo por segundo.

El costo de una campaña en diarios o revistas se calcula como el producto entre la cantidad de centímetros del aviso, un monto fijo por centímetro y la cantidad de días que dura la campaña.

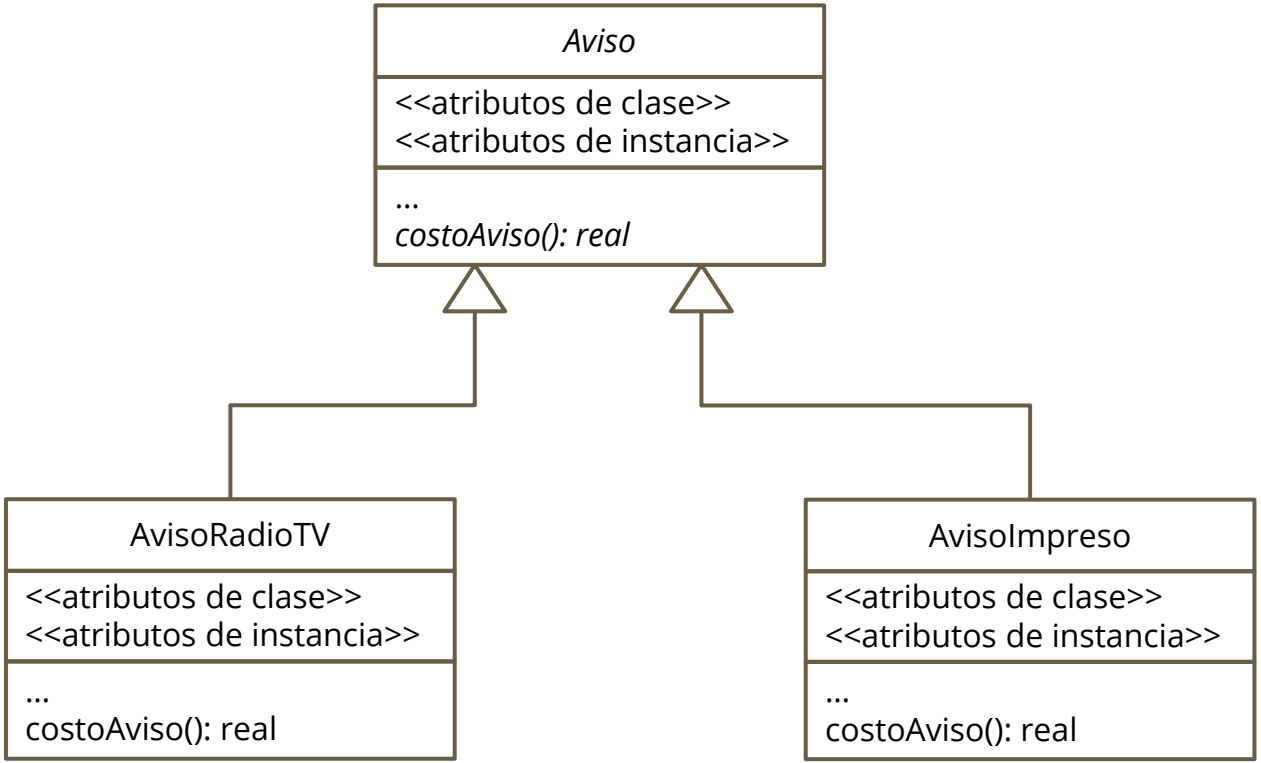
Caso de estudio: Agencia Publicitaria

En el caso de estudio descrito un aviso concreto se publica en radio o en televisión, y también podría publicarse en revistas o diarios.

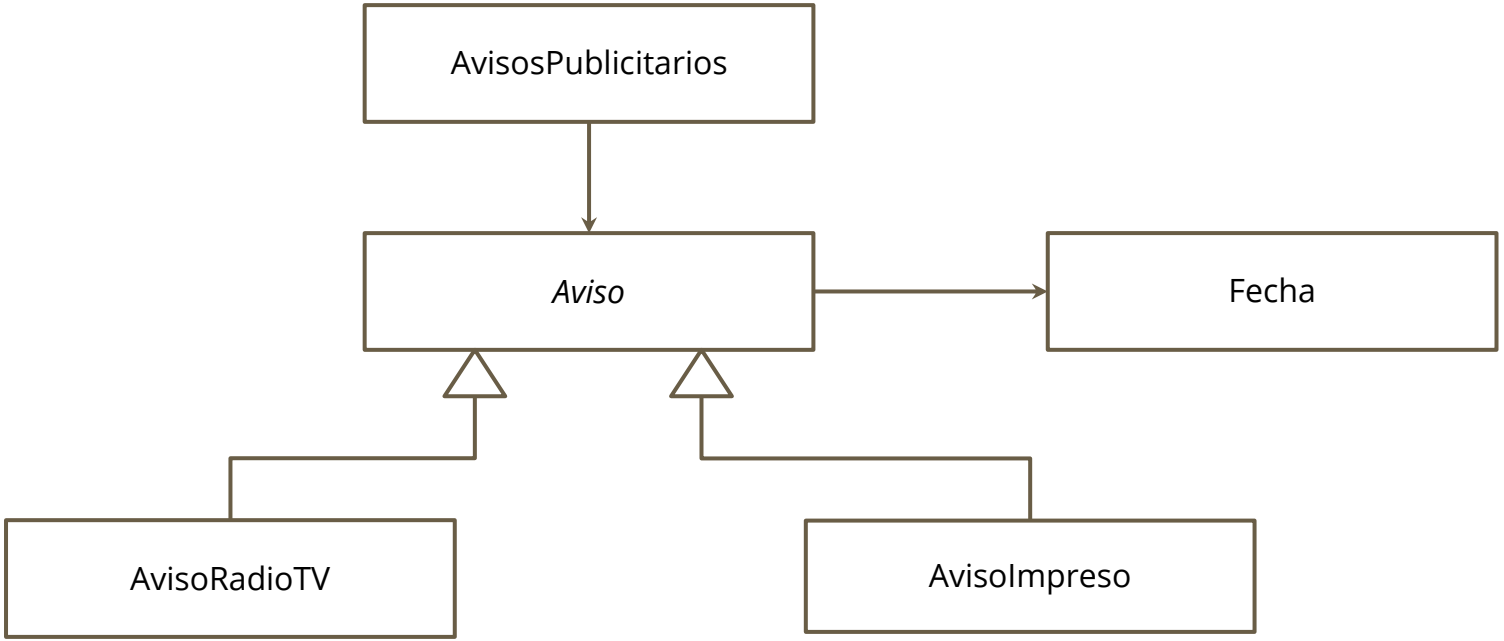
De modo que podemos definir una clase **Aviso** para factorizar atributos y comportamiento compartido. Esta clase no modela ningún objeto del problema real, **no tiene sentido crear objetos de software de esta clase**.

El método **costoAviso()** es abstracto, todos los avisos tienen un costo pero la manera de calcularlo depende del medio en el que se publica.

Caso de estudio: Agencia Publicitaria



Caso de estudio: Agencia Publicitaria



Caso de estudio: Agencia Publicitaria

```python
from abc import ABC, abstractmethod
from Fecha import Fecha

class Aviso(ABC):
    def __init__(self, nombre:str, producto:str, empresa:str, fecha:Fecha, cantDias:int):
        if not isinstance(nombre, str) or nombre == "" or nombre.isspace():
            raise ValueError("El nombre debe ser un string válido.")
        if not isinstance(producto, str) or producto == "" or producto.isspace():
            raise ValueError("El producto debe ser un string válido.")
        if not isinstance(empresa, str) or empresa == "" or empresa.isspace():
            raise ValueError("La empresa debe ser un string válido.")
        if not isinstance(fecha, Fecha):
            raise ValueError("La fecha debe ser un objeto de tipo Fecha.")
        if not isinstance(cantDias, int) or cantDias <= 0:
            raise ValueError("La cantidad de días debe ser un número entero positivo.")
        self._nombre = nombre
        self._producto = producto
        self._empresa = empresa
        self._fecha = fecha
        self._cantDias = cantDias

    @abstractmethod
    def costoAviso()->float:
        pass
```

Caso de estudio: Agencia Publicitaria

Como no existen instancias de una clase abstracta, el constructor no va a ser invocado explícitamente para crear objetos de la clase, sino desde los constructores de las clases derivadas para asegurarse de que la lógica de inicialización de la clase abstracta sea ejecutada.

El atributo cantDias indica la cantidad de días que dura la campaña.

No hay dos avisos que coincidan en los atributos nombre-empresa, es decir, puede haber varios avisos de la misma empresa O con el mismo nombre, pero no con de la misma empresa Y con el mismo nombre.

Caso de estudio: Agencia Publicitaria

```
class AvisoRadioTV(Aviso):
    def __init__(self, nombre:str, producto:str, empresa:str, fecha:Fecha, cantDias:int,
                 duracion:int, repeticionesPorDia:int, montoPorSegundo:float):
        super().__init__(nombre, producto, empresa, fecha, cantDias)
        if not isinstance(duracion, int) or duracion <= 0:
            raise ValueError("La duración debe ser un número entero positivo.")
        if not isinstance(repeticionesPorDia, int) or repeticionesPorDia <= 0:
            raise ValueError("La cantidad de repeticiones por día debe ser un número entero positivo.")
        if not isinstance(montoPorSegundo, (int, float)) or montoPorSegundo <= 0:
            raise ValueError("El monto por segundo debe ser un número positivo.")
        self.__duracion = duracion
        self.__repeticionesPorDia = repeticionesPorDia
        self.__montoPorSegundo = montoPorSegundo

    def costoAviso(self)->float:
        return self.__duracion * self.__montoPorSegundo * self.__repeticionesPorDia * self._cantDias
```

Caso de estudio: Agencia Publicitaria

```
class AvisoImpreso(Aviso):
    def __init__(self, nombre:str, producto:str, empresa:str, fecha:Fecha, cantDias:int,
                 centCentimetros:int, costoPorCentimetro:float):
        super().__init__(nombre, producto, empresa, fecha, cantDias)
        if not isinstance(centCentimetros, int) or centCentimetros <= 0:
            raise ValueError("La cantidad de centímetros debe ser un número entero positivo.")
        if not isinstance(costoPorCentimetro, (int, float)) or costoPorCentimetro <= 0:
            raise ValueError("El costo por centímetro debe ser un número positivo.")
        self.__centCentimetros = centCentimetros
        self.__costoPorCentimetro = costoPorCentimetro

    def costoAviso(self)->float:
        return self.__centCentimetros * self.__costoPorCentimetro * self._cantDias
```

Caso de estudio: Agencia Publicitaria

La clase AvisosPublicitarios encapsula una colección de elementos de clase Aviso, representada con una lista (arreglo).

La clase brinda servicios para:

- Insertar un nuevo Aviso, debe verificar que no exista un Aviso con la misma clave y el aviso no sea nulo.
- Eliminar un Aviso
- Decidir si existe un aviso con una **clave dada**.

Caso de estudio: Agencia Publicitaria

```
class AvisosPublicitarios:
    def __init__(self):
        self.__listaAvisos = []

    def obtenerAvisos(self)->list[Aviso]:
        return self.__listaAvisos

    def existeAviso(self, nombre:str, empresa:str)->bool:
        for aviso in self.__listaAvisos:
            if aviso.obtenerNombre() == nombre and aviso.obtenerEmpresa() == empresa:
                return True
        return False

    def agregarAviso(self, aviso:Aviso):
        if not isinstance(aviso, Aviso):
            raise ValueError("El aviso debe ser un objeto de tipo Aviso.")
        if not self.existeAviso(aviso.obtenerNombre(), aviso.obtenerEmpresa()):
            self.__listaAvisos.append(aviso)
        else:
            raise ValueError("Ya existe un aviso con ese nombre y empresa.")
```

Caso de estudio: Agencia Publicitaria

```
def eliminarAviso(self, nombre:str, empresa:str):
    if not isinstance(nombre, str) or nombre == "" or nombre.isspace():
        raise ValueError("El nombre debe ser un string válido.")
    if not isinstance(empresa, str) or empresa == "" or empresa.isspace():
        raise ValueError("La empresa debe ser un string válido.")
    if self.existeAviso(nombre, empresa):
        for aviso in self.__listaAvisos:
            if aviso.obtenerNombre() == nombre and aviso.obtenerEmpresa() == empresa:
                self.__listaAvisos.remove(aviso)
    else:
        raise ValueError("No se encontró el aviso a eliminar.")

def costoTotalAvisos(self)->float:
    total = 0
    for aviso in self.__listaAvisos:
        total += aviso.costoAviso()
    return total
```

Caso de estudio: Agencia Publicitaria

```
class AvisoRadioTV(Aviso):

    def costoAviso(self)->float:
        return self.__duracion * self.__montoPorSegundo * self.__repeticionesPorDia * self._cantDias
```

Si en las subclases los atributos se acceden indirectamente a través de los servicios, la implementación de la clase base puede cambiar y el cambio no afecta a las clases clientes.

```
class AvisoRadioTV(Aviso):

    def costoAviso(self)->float:
        return self.__duracion * self.__montoPorSegundo * self.__repeticionesPorDia * self.obtenerCantDias()
```

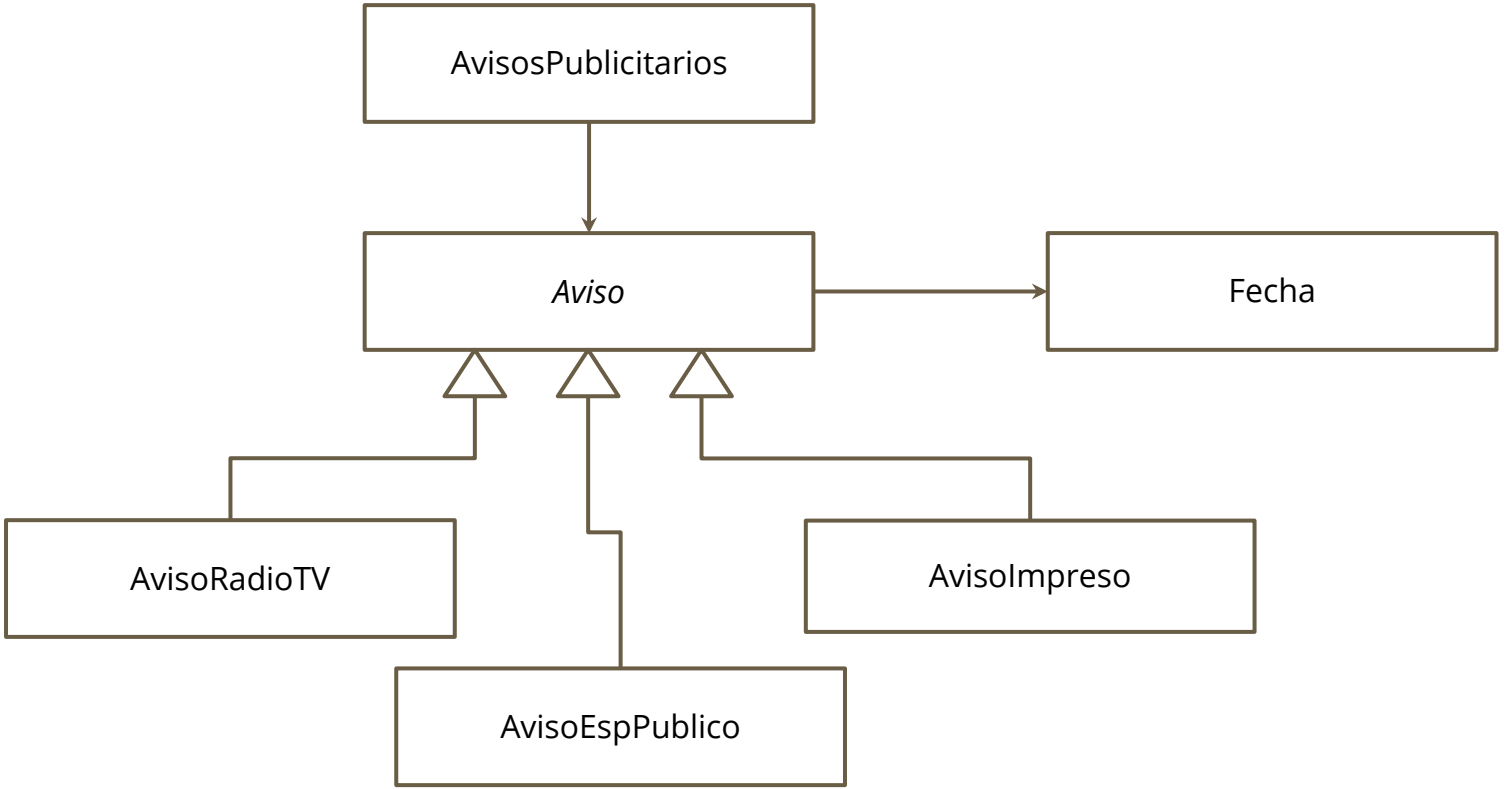
Caso de estudio: Agencia Publicitaria

Consideremos que el diseño del problema tiene que extenderse para incluir también avisos en espacios públicos que administra la Municipalidad. La Municipalidad tiene codificados los espacios públicos con un código numérico que los identifica y ofrece tres tipos de carteles para colocar en esos espacios a los que denomina A, B y C.

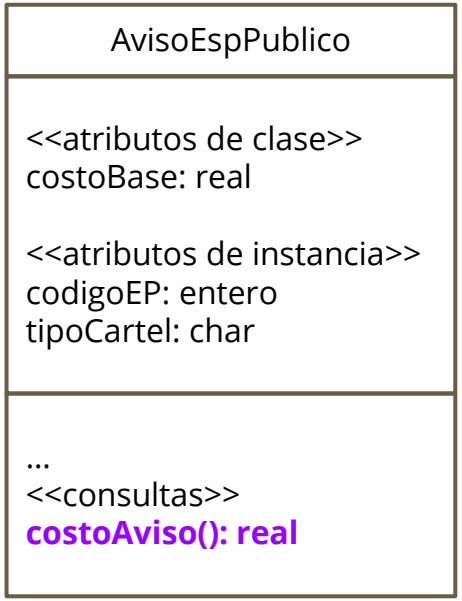
De modo que la clase AvisoEspPublico tiene dos atributos codigoEP (entero) y tipoCartel (caracter).

El costo de un aviso en un espacio público depende del tipo de cartel; el costo del cartel de tipo B es el doble que el de tipo A (que es una constante) y el C el doble que el B.

Caso de estudio: Agencia Publicitaria

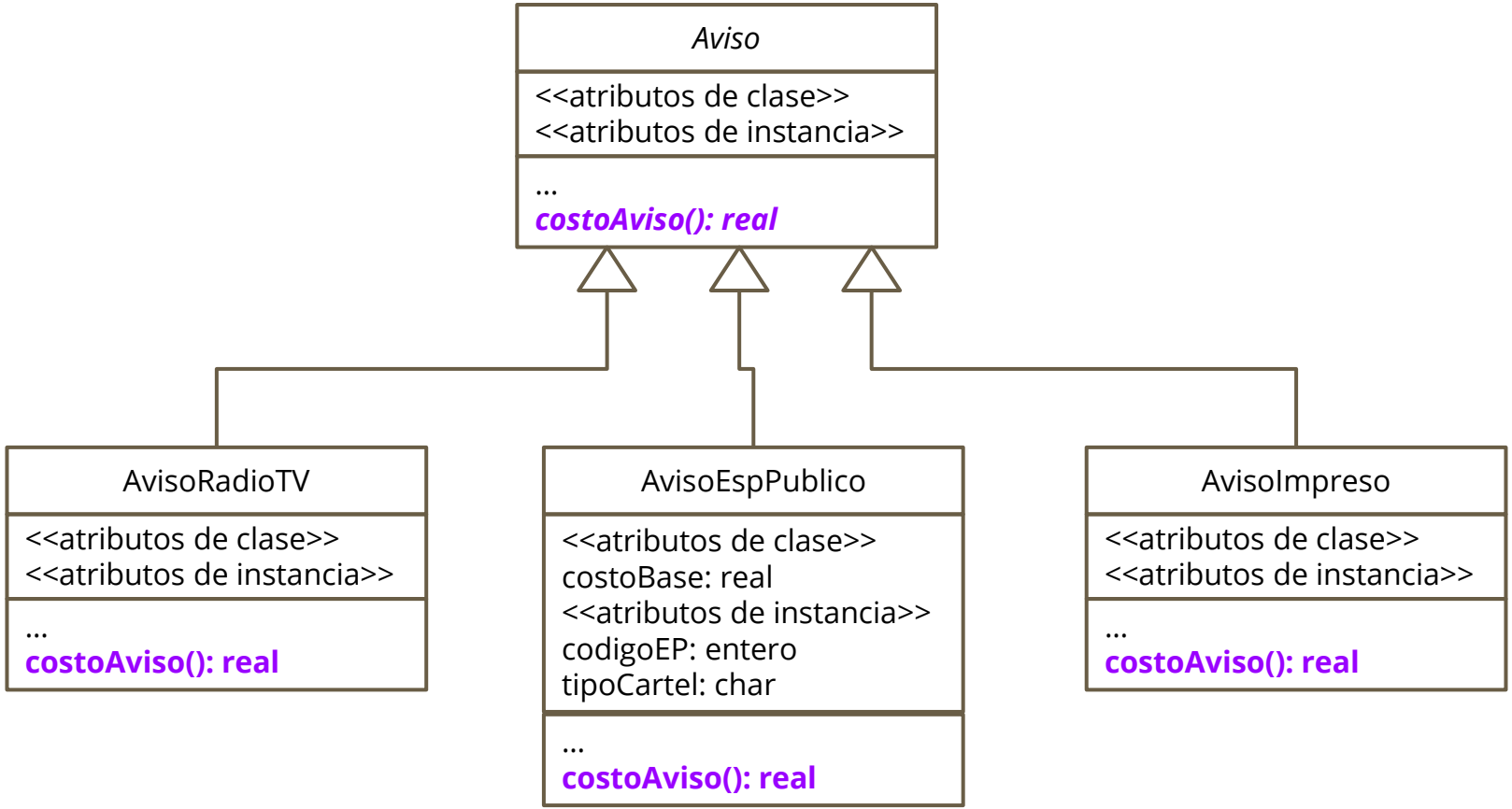


Caso de estudio: Agencia Publicitaria



costoAviso() es un método polimórfico (al igual que en las otras subclases, en cada clase tiene un comportamiento particular). La implementación de estos métodos varía según el tipo de aviso, cada clase proporciona su implementación.

Caso de estudio: Agencia Publicitaria

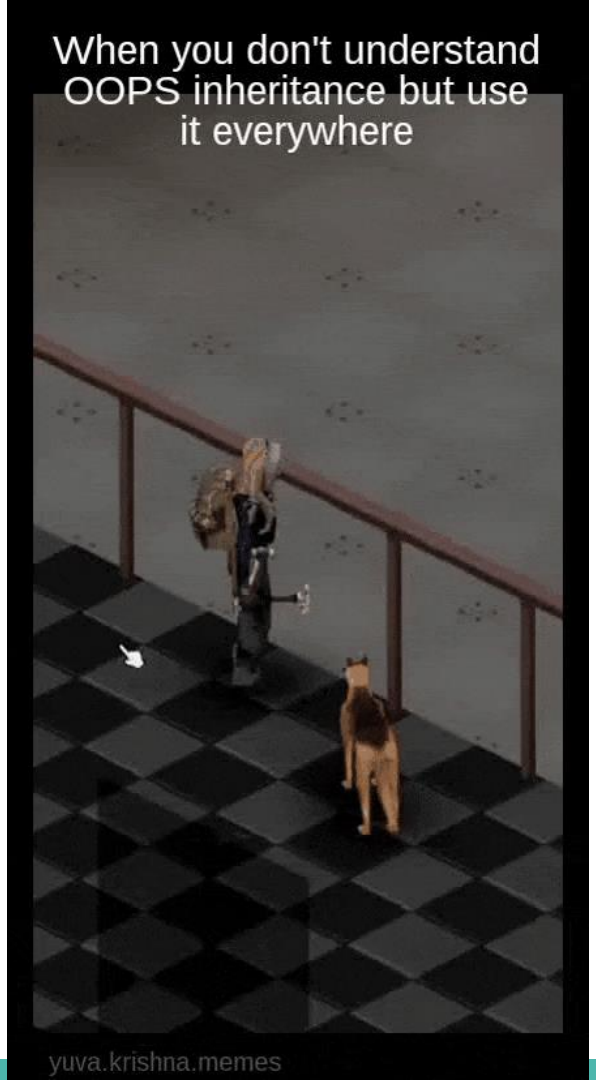


Caso de estudio: Agencia Publicitaria

El cambio en la especificación no afecta a las clases **Aviso**, **AvisoImpreso**, **AvisoRadioTV** y **AvisosPublicitarios**.

```
class AvisosPublicitarios:

    def costoTotalAvisos(self)->float:
        total = 0
        for aviso in self.__listaAvisos:
            total += aviso.costoAviso()
        return total
```


When you don't understand OOPS inheritance but use it everywhere
yuva.krishna.memes