POO: ¿Qué es la programación orientada a objetos?

João Henrique
10/11/2020

Programación orientada a objetos y programación estructurada

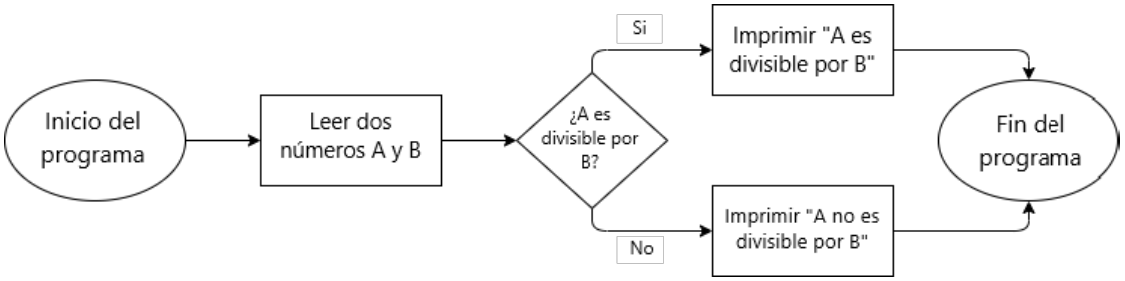
Como la mayoría de las actividades que hacemos a diario, la programación también tiene diferentes formas de realizarse. Estos modos se llaman **paradigmas de programación** y entre ellos están la **programación orientada a objetos** (POO) y la programación estructurada. Cuando comenzamos a usar lenguajes como Java, C#, Python y otros que posibilitan el paradigma orientado a objetos, es común cometer errores y aplicar la programación estructurada pensando que estamos usando recursos de la orientación a objetos.

En la **programación estructurada**, un programa consta de tres tipos básicos de estructuras:

- Secuencias: son los comandos a ejecutar
- Condiciones: secuencias que solo deben ejecutarse si se cumple una condición (ejemplos: `if-else`, `switch` y comandos similares)
- Repeticiones: secuencias que deben realizarse repetidamente hasta que se cumpla una condición (`for`, `while`, `do-while`, etc.)

Estas estructuras se utilizan para procesar la entrada del programa, cambiando los datos hasta que se genera la salida esperada. Hasta ahora, nada que la programación orientada a objetos no haga también, ¿verdad?

La principal diferencia es que, en la programación estructurada, un programa generalmente se escribe en **una sola rutina** (o función) y, por supuesto, puede dividirse en subrutinas. Pero el flujo del programa sigue siendo el mismo, como si se pudiese copiar y pegar el código de las subrutinas directamente en las rutinas que las llaman, de tal forma que, al final, solo existiese una gran rutina que ejecute todo el programa.



Además, el acceso a las variables no tiene muchas restricciones en la programación estructurada. En lenguajes fuertemente basados en este paradigma, restringir el acceso a una variable se limita a decir si es visible o no dentro de una función (o módulo, como en el uso de la palabra clave `static`, en lenguaje C), pero no es posible decir de forma nativa que solo se puede acceder a una variable mediante unas pocas rutinas del programa. El esquema para situaciones como estas implica prácticas de programación

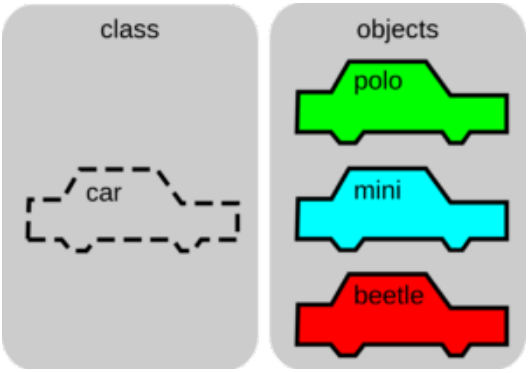
perjudiciales para el desarrollo del sistema, como el **uso excesivo de variables globales**. Vale la pena recordar que las variables globales se usan típicamente para mantener estados en el programa, marcando en qué parte de la ejecución se encuentran.
La **programación orientada a objetos** surgió como una alternativa a estas características de la programación estructurada. El propósito de su creación fue también acercar el manejo de las estructuras de un programa al manejo de las cosas en el mundo real, de ahí el nombre "objeto" como algo genérico, que puede representar cualquier cosa tangible.

Este nuevo paradigma se basa principalmente en dos conceptos clave: **clases** y **objetos**. Todos los demás conceptos, igualmente importantes, se basan en estos dos.

¿Qué son clases y objetos?

Imagina que recientemente compraste un auto y decidiste **modelar ese auto utilizando programación orientada a objetos**. Tu auto tiene las características que estabas buscando: motor 2.0 híbrido, azul oscuro, cuatro puertas, cambio automático, etc. También tiene comportamientos que probablemente fueron el motivo de tu compra, como acelerar, reducir la velocidad, encender los faros, tocar la bocina y tocar música. Podemos decir que el auto nuevo es un *objeto*, donde sus características son sus *atributos* (datos vinculados al objeto) y sus comportamientos son acciones o *métodos*.

Tu auto es un objeto tuyo, pero en la tienda donde lo compraste habían otros tantos, muy similares, con cuatro ruedas, volante, cambio, espejos retrovisores, faros, entre otras partes. Ten en cuenta que, aunque tu auto es único (por ejemplo, tiene un registro único en el Departamento de Tránsito), puede haber otros con exactamente los mismos atributos, o similares, o incluso totalmente diferentes, pero que aún se consideran *autos*. Entonces podemos decir que tu objeto se puede clasificar (es decir, tu *objeto pertenece a una clase*) como un auto, y que tu auto no es más que una instancia de esa clase llamada "auto".



Así, abstrayendo un poco la analogía, **una clase es un conjunto de características y comportamientos que definen el conjunto de objetos pertenecientes a esta clase**. Tenga en cuenta que la clase en sí es un concepto abstracto, como un molde, que se vuelve concreto y palpable a través de la creación de un objeto. Llamamos a esta creación de *instanciación de clase*, como si estuviéramos usando este molde (clase) para crear un objeto.

Ejemplo de Java

```
public class Auto {
```

```
    Double velocidad;
    String modelo;

    public Auto(String modelo) {
        this.modelo = modelo;
        this.velocidad = 0;
    }

    public void acelerar() {
        /* código del auto para acelerar */
    }

    public void frear() {
        /* código del auto para frenar */
    }

    public void acenderFaro() {
        /* código del auto para encender el faro */
    }
}
```

Ejemplo en Python

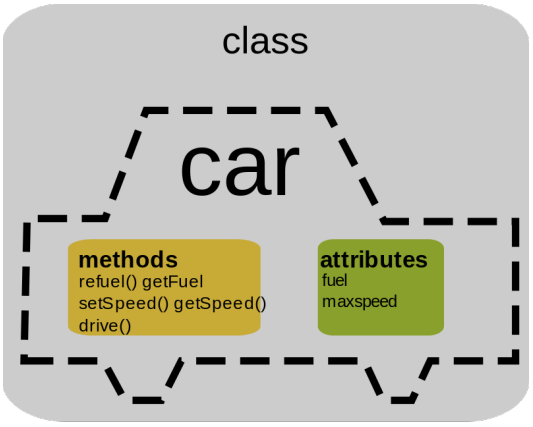
```
  class Auto:
      def __init__(self, modelo):
          self.modelo = modelo;
          self.velocidad = 0

      def acelerar(self):
          # Codigo para acelerar el auto

      def frear(self):
          # Codigo para frenar el auto

      def encenderFaro(self):
          # Codigo para encender el faro del auto
```

Encapsulamiento, herencia y polimorfismo: las principales características de la POO

Las dos bases de la POO son los conceptos de **clase y objeto**. De estos conceptos derivan algunos otros conceptos extremadamente importantes al paradigma, que no solo lo definen, sino que son las soluciones a algunos problemas de la programación estructurada. Los conceptos en cuestión son el *encapsulamiento*, la *herencia*, las *interfaces* y el *polimorfismo*.

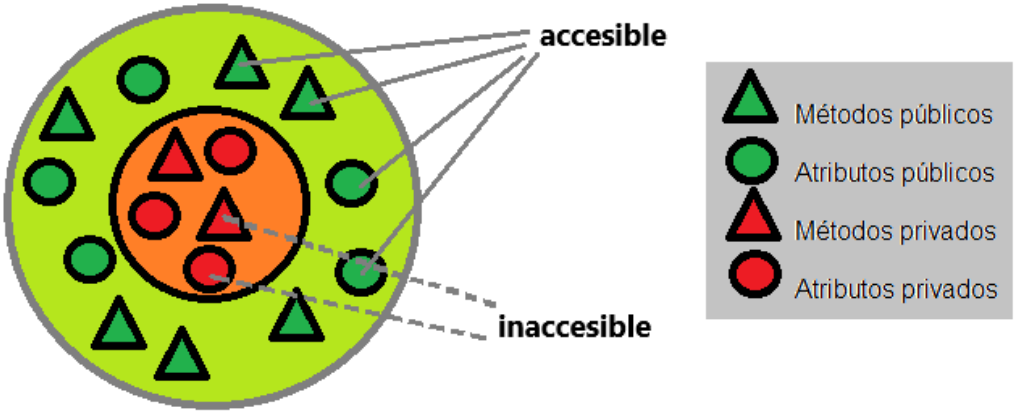
Encapsulamiento

Aun usando la analogía del auto, sabemos que tiene **atributos y métodos**, es decir, características y comportamientos. Los métodos del auto, como acelerar, pueden usar atributos y otros métodos del auto, como el tanque de gasolina y el mecanismo de inyección de combustible, respectivamente, ya que acelerar consume combustible.



Sin embargo, si algunos de estos atributos o métodos son fácilmente visibles y modificables, como el mecanismo de aceleración del auto, esto puede darle libertad para realizar cambios, lo que resulta en efectos secundarios imprevisibles. En esta analogía, una persona puede no estar satisfecha con la aceleración del auto y cambia la forma en que se produce, creando efectos secundarios que pueden hacer incluso con que el auto ni ande, por ejemplo.

En este caso, decimos que el método de aceleración de su auto no es visible desde el exterior del mismo auto. En la POO, un atributo o método que no es visible desde afuera del mismo objeto se llama "**privado**" y cuando está visible, se llama "**público**".



Pero entonces, ¿cómo sabemos cómo acelera nuestro auto? Es simple: no lo sabemos. Solo sabemos que para acelerar hay que pisar el acelerador y el objeto sabe cómo realizar esta acción sin exponer cómo lo hace. Decimos que la aceleración del auto está **encapsulada**, porque sabemos lo que hará cuando ejecutemos este método, pero no sabemos cómo y, de hecho, al programa **no le importa cómo lo hace el objeto, solo que lo haga**.

Lo mismo ocurre con los atributos. Por ejemplo: no sabemos cómo el auto sabe qué velocidad mostrar en el velocímetro o cómo calcula su velocidad, pero no necesitamos saber cómo se hace eso. Solo necesitamos saber que nos dará la velocidad adecuada. Se puede leer o cambiar un atributo encapsulado desde ***getters*** **y** ***setters***.

Este encapsulamiento de atributos y métodos evita la llamada *fuga de alcance*, donde un atributo o método es visible para alguien que no debería poder verlo, como otro objeto o clase. Esto evita confundir el uso de variables globales en el programa, facilitando identificar en qué estado estará cada variable en cada momento del programa, ya que la restricción de acceso nos permite identificar quién puede modificarla.

Ejemplo en Java

```
public class Auto {
    private Double velocidad;
    private String modelo;
    private MecanismoAceleracion mecanismoAceleracion;
    private String color;

    /* Vea que se inserta el mecanismo de aceleración en el auto al construirse, y
        no lo vemos ni podemos modificarlo, es decir, no tiene getter ni setter.
        Ya el "modelo" se puede ver, pero no se puede alterar. */
    public Auto(String modelo, MecanismoAceleracion mecanismoAceleracion) {
        this.modelo = modelo;
        this.mecanismoAceleracion = mecanismoAceleracion;
        this.velocidad = 0;
    }

    public void acelerar() {
        this.mecanismoAceleracion.acelerar();
    }

    public void frenar() {
        /* código del auto para frenar */
    }

    public void encenderFaro() {
        /* código del auto para encender el faro */
    }

    public Double getVelocidad() {
        return this.velocidad
    }

    private void setVelocidad() {
        /* código para alterar la velocidad del auto */
        /* Como solo el propio auto debe calcular la velocidad,
            este método no se puede llamado de afuera, por eso es "private" */
    }
```

```
    public String getModelo() {
        return this.modelo;
    }

    public String getCor() {
        return this.cor;
    }

    /* podemos cambiar el color del auto cuando deseemos */
    public void setCor(String color) {
        this.cor = color;
    }
}
```

Ejemplo en Python

```
# Ejemplo de la clase Auto en Python
class Auto:
    def __init__(self, modelo, mecanismoAceleracion):
        self.__modelo = modelo;
        self.__velocidad = 0
        self.__mecanismoAceleracion = mecanismoAceleracion

    def acelerar(self):
        mecanismoAceleracion.acelerar()

    def frear(self):
        # Codigo para frenar el auto

    def encenderFaro(self):
        # Codigo para encender el faro del auto

    def getVelocidad(self):
        return self.velocidad

    def __setVelocidad(self):
        # Codigo para alterar la velocidad por dentro del objeto

    def getModelo(self):
        return self.modelo

    def getColor(self):
        return self.color
```

```
        def setColor(self, color):
            self.color = color
```

Herencia

En nuestro ejemplo, acabas de comprar un auto con los atributos que buscabas. A pesar de ser únicos, hay autos con **exactamente los mismos atributos o formas modificadas**. Digamos que compraste el modelo Fit de Honda. Este modelo tiene otra versión llamada WR-V (o "Honda Fit Cross Style"), que tiene muchos de los atributos de la versión clásica, pero con algunas diferencias muy grandes para transitar por caminos de tierra: el motor es híbrido (acepta alcohol y gasolina), tiene un sistema de suspensión diferente, y supongamos que además tenga un sistema de tracción diferente (tracción a las cuatro ruedas, por ejemplo). Vemos entonces que no solo cambian algunos atributos sino también algunos mecanismos (o métodos, traduciéndose a POO), pero esta versión "cross" sigue siendo el modelo Honda Fit, es decir, es *un tipo* del modelo.

Cuando decimos que **una clase A es *un tipo de* clase B**, decimos que clase A *hereda* las características de la clase B y que la clase B es *madre* de la clase A, estableciendo entonces una relación de herencia entre ellas. En el caso del auto, decimos que un Honda Fit "Cross" es *un tipo de* Honda Fit, y lo que cambia son algunos atributos (tapabarros reforzado, altura de la suspensión, etc.), y uno de los métodos de la clase (acelerar, ya que ahora hay tracción en las cuatro ruedas), pero todo lo demás sigue igual, y el nuevo modelo recibe los mismos atributos y métodos que el modelo clásico.

Ejemplo en Java

```
// "extends" establece la relacion de herencia con la clase Auto
public class HondaFit extends Auto {

    public HondaFit(MecanismoAceleracion mecanismoAceleracion) {
        String modelo = "Honda Fit";
        // llama al constructor de la clase madre, es decira, de la clase "Auto"
        super(modelo, mecanismoAceleracion);
    }
}
```

Ejemplo en Python

```
# Las clases dentro del parénteses son las clases madre de la clase definiendose
class HondaFit(Auto):

    def __init__(self, mecanismoAceleracion):
        modelo = "Honda Fit"
        # llama al constructor de la clase madre, es decir, de la clase "Auto"
        super().__init__(self, modelo, mecanismoAceleracion)
```

Interfaz

Muchos de los métodos de los autos son comunes en muchos automóviles. Tanto un auto como una motocicleta son clases cuyos objetos pueden acelerar, frenar, encender los faros, etc., ya que son comunes a los automóviles. Podemos decir, entonces, que tanto la clase "auto" como la "motocicleta" *son* "automóviles".

Cuando dos (o más) clases tienen comportamientos comunes que pueden separarse en otra clase, decimos que **la "clase común" es una *interfaz*, que puede ser "heredada" por las otras clases**. Tenga en cuenta que colocamos la interfaz como una "clase común", que puede ser "heredada" (con comillas), porque una interfaz no es exactamente una clase, sino un conjunto de métodos que todas las clases que heredan de ella deben tener (implementar) - por lo tanto, una interfaz no es "heredada" por una clase, sino *implementada*. En el mundo del desarrollo de software, decimos que una interfaz es un "contrato": una clase que implementa una interfaz debe proporcionar una implementación para **todos** los métodos que define la interfaz y, a cambio, la clase de implementación puede decir que es del tipo de interfaz. En nuestro ejemplo, "auto" y "motocicleta" son clases que implementan los métodos de la interfaz "automóvil", por lo que podemos decir que cualquier objeto de estas dos primeras clases, como una Honda Fit o una motocicleta Yamaha, son automóviles.

Un pequeño detalle: **una interfaz no puede ser heredada por una clase**, sino implementada. Sin embargo, **una interfaz puede heredar de otra interfaz**, creando una jerarquía de interfaces. Usando un ejemplo completo con autos, decimos que la clase "Honda Fit Cross" hereda de la clase "Honda Fit", que a su vez hereda de la clase "Auto". La clase "Auto" implementa la interfaz "Automóvil" que, a su vez, puede heredar (por ejemplo) una interfaz llamada "MedioDeTransporte", ya que tanto un "automóvil" como un "carro" son medios de transporte, aunque un carro no sea un automóvil.

Ejemplo en Java

```
public interface Automovil {
    void acelerar();
    void frenar();
    void encenderFaro();
}

public class Auto implements Automovil {

    /* ... */

    @Override
    public void acelerar() {
        this.mecanismoAceleracion.acelerar();
    }

    @Override
    public void frenar() {
```

```
        /* código del auto para frenar */
    }

    @Override
    public void encenderFaro() {
        /* código del auto para encender el faro */
    }

    /* ... */
}

public class Moto implements Automovil {

    /* ... */

    @Override
    public void acelerar() {
        /* código específico de la moto para acelerar */
    }

    @Override
    public void frenar() {
        /* código específico de la moto para frenar */
    }

    @Override
    public void encenderFaro() {
        /* código específico de la moto para encender el faro */
    }

    /* ... */
}
```

Ejemplo en Python

```
class Automovil():
    def acelerar(self):
        raise NotImplementedError()

    def frenar(self):
        raise NotImplementedError()

    def encenderFarol(self):
        raise NotImplementedError()
```

```
class Auto(Automovil):

    # ...

    def acelerar(self):
        # Codigo para acelerar el auto

    def frenar(self):
        # Codigo para frenar el auto

    def encenderFaro(self):
        # Codigo para encender el faro del auto

    # ...

class Moto(Automovil):

    # ...

    def acelerar(self):
        # Codigo para acelerar la moto

    def frear(self):
        # Codigo para frenar la moto

    def encenderFaro(self):
        # Codigo para encender la moto

    # ...
```

Nota: crea un error de tipo `NotImplementedError`, aquello, evidencia que, si una clase hija intenta ejecutar un método de la clase padre sin haberlo implementado, se produce el error. En Python, las interfaces se crean como clases normales que heredan las clases hijas. Hay formas de forzar la implementación por parte de las clases hijas, pero para nuestro ejemplo, este abordaje es suficiente.

Polimorfismo

Digamos que una de las razones por las que compraste un auto fue la calidad de su sistema de sonido. Pero, en su caso, digamos que la reproducción solo se puede hacer por radio o *bluetooth*, mientras que en su auto antiguo, solo se puede hacer a través de la tarjeta SD y *pendrive*. El método de "reproducir música" está presente en ambos autos, pero como su sistema de sonido es diferente, la forma en que el automóvil reproduce música es diferente. Decimos que el método "reproducir música" es una forma de **polimorfismo**, porque dos objetos, de dos clases diferentes, tienen **el mismo método que se implementa**

de diferentes maneras, es decir, un método tiene *varias formas*, varias implementaciones diferentes en diferentes clases, pero que tienen el mismo efecto ("polimorfismo" proviene del griego *poli* = muchas, *morfos*= forma).

Ejemplo en Java

```
lic class Main {
 public static void main(String[] args) {
     Automovil moto = new Moto("Yamaha XPTO-100", new MecanismoDeAceleracionDeMotos
     Automovil auto = new Auto("Honda Fit", new MecanismoDeAceleracionDeAutos())
     List<Automovil> listaAutomoviles = Arrays.asList(moto, auto);
     for (Automovil automovil : listaAutomoviles) {
         automovil.acelerar();
         automovil.encenderFaro();
     }
 }
```

Ejemplo en Python

```
def main():
    moto = Moto("Yahama XPTO-100", MecanismoDeAceleracionDeMotos())
    auto = Auto("Honda Fit", MecanismoDeAceleracionDeAutos())
    listaAutomoviles = [moto, auto]
    for automovil in listaAutomoviles:
        automovil.acelerar()
        automovil.encenderFaro()
```

Ten en cuenta que, aunque son objetos diferentes, `moto` y `auto` tienen los mismos métodos `acelerar` y `encenderFaro`, que se llaman igual, a pesar de estar implementados de manera diferente.

Design Patterns

Algunos problemas aparecen con tanta frecuencia en **POO** que sus soluciones se han convertido en estándares de diseño de sistemas y modelado de código orientado a objetos para solucionarlos. Estos **estándares de proyecto**, (o **design patterns**) no son más que formas estandarizadas de solucionar problemas comunes en lenguajes orientados a objetos. El libro "Design Patterns", conocido como **Gof:Gang of Four**, es la principal referencia en este tema, conteniendo los principales estándares utilizados en grandes proyectos.

Clean code y SOLID

En proyectos desarrollados con POO, como en cualquier otro, el código puede volverse desordenado y difícil de mantener a medio y largo plazo. Ante esta situación, se han desarrollado algunos principios de buenas prácticas de programación y código limpio, por ejemplo:

- KISS (*Keep It Simple, Stupid*, "Mantén las cosas sencillas"): Siempre que se escribe un código, debe escribirse de la manera más sencilla posible, para mantener el código más legible. Códigos demasiado complejos son más difíciles de mantener, ya que es más difícil entender qué hace y cómo lo hace.
- DRY (*Don't Repeat Yourself*, "No repita"): Todo el código escrito para solucionar un problema debe escribirse solo una vez, para evitar la repetición del código. Es casi una variación de KISS, ya que la repetición del código lo hace más confuso y difícil de mantener y corregir, si es necesario.

Además de los **design patterns** y de los principios del código limpio existe un conjunto de técnicas, más generalizadas que los design patterns, que ayudan a crear código orientado a objeto para hacerlo más maleable, permitiendo un mantenimiento y expansión del código más fluido y sin complicaciones a lo largo del tiempo.