El paradigma orientado a objetos

Programación Orientada a Objetos (POO): Conceptos Fundamentales

El paradigma de la **Programación Orientada a Objetos (POO)**, del inglés *Object-Oriented Programming (OOP)*, es un enfoque de programación que se fundamenta en el concepto de **objeto**. Un objeto es una entidad que encapsula datos y los procedimientos que operan sobre esos datos, denominados **métodos**, en una única unidad cohesionada.

Distinción con Paradigmas Imperativos

A diferencia de los paradigmas imperativos, donde los datos se almacenan en variables y su manipulación se realiza mediante funciones independientes, la POO integra la información y el comportamiento dentro de la misma estructura del objeto. Esta integración eleva el nivel de abstracción y favorece la modularidad en el desarrollo de software.

Principios Clave de la POO

La POO se asienta sobre pilares conceptuales que facilitan el diseño, la implementación y el mantenimiento de sistemas complejos:

- **Encapsulación:** Este principio consiste en la agrupación de los datos (atributos) y los métodos que operan sobre ellos dentro de una única unidad, el objeto. La encapsulación restringe el

acceso directo a los datos internos de un objeto, obligando a la interacción a través de una **Interfaz de Programación de Aplicaciones (API)** pública o conjunto de métodos expuestos. Esto protege la integridad de los datos, previene modificaciones inconsistentes y facilita la evolución del código sin afectar a otras partes del sistema que interactúan con el objeto.

- **Abstracción:** Implica la representación simplificada de entidades del mundo real o conceptos complejos, enfocándose en sus características esenciales y ocultando los detalles de implementación subyacentes. La POO permite a los desarrolladores trabajar con objetos a un nivel conceptual, sin necesidad de comprender su funcionamiento interno detallado, facilitando así la gestión de la complejidad.
- **Polimorfismo:** Del griego "muchas formas", el polimorfismo permite que un mismo nombre de método o mensaje pueda invocar diferentes implementaciones según el tipo específico de objeto que lo reciba. Esto conduce a un código más flexible, extensible y reutilizable, ya que se puede interactuar con diferentes objetos de manera uniforme a través de una interfaz común.
- **Herencia:** En sistemas basados en **clases**, la herencia es un mecanismo que permite a una clase (subclase o clase derivada) adquirir propiedades y comportamientos (atributos y métodos) de otra clase (superclase o clase base). Este principio promueve la **reutilización de código** y la organización jerárquica de los objetos, modelando relaciones "es un tipo de" (por ejemplo, un "Coche" es un tipo de "Vehículo").

Clases y Prototipos

La creación y estructuración de objetos en la POO se puede abordar de dos formas principales:

- **Clases:** Son plantillas o moldes que definen la estructura y el comportamiento común de un conjunto de objetos. Un objeto creado a partir de una clase se denomina **instancia** de esa clase. La mayoría de los lenguajes de POO (como Java, C++, Python, C#) se basan en clases y en el concepto de herencia para la creación de jerarquías de tipos.
- **Prototipos:** En algunos lenguajes (notablemente JavaScript), los objetos se crean directamente a partir de otros objetos existentes, denominados prototipos. En este modelo, un objeto puede servir como plantilla para la creación de nuevos objetos, los cuales heredan directamente las propiedades y métodos del prototipo.

> **Actividad**
> De los siguientes ejemplos del mundo real, separa los datos de los métodos. Esta habilidad es crucial a la hora de modelar objetos a partir de especificaciones.
>
> 1. Un Turismo (Coche)
> 2. Un Tucán (pájaro)
> 3. Una profesora de informática (Persona)

Principios SOLID

Los **principios SOLID** son un conjunto de cinco principios de diseño de software que tienen como objetivo hacer que los diseños de software sean más comprensibles, flexibles y fáciles de mantener. Fueron introducidos por Robert C. Martin, también conocido como "Uncle Bob", y son una guía fundamental en la programación orientada a objetos para escribir código limpio y robusto. Al aplicarlos, se busca crear sistemas que sean más resistentes a los cambios y menos propensos a errores.

S - Principio de Responsabilidad Única (Single Responsibility Principle - SRP)

El SRP establece que **una clase debe tener una, y solo una, razón para cambiar**. Esto significa que una clase debe ser responsable de una única funcionalidad o de un único aspecto de la aplicación. Si una clase tiene múltiples responsabilidades, cualquier cambio en una de esas responsabilidades podría afectar a las demás, haciendo que el código sea más frágil y difícil de mantener.

Ejemplo: En lugar de tener una clase `Reporte` que se encargue tanto de generar los datos del informe como de formatearlos para su impresión, sería mejor tener dos clases separadas: una `GeneradorDeDatosDeReporte` y una `FormateadorDeReporte`. Así, si cambian las reglas de negocio para los datos, solo se modifica la primera; si cambia el diseño de impresión, solo se modifica la segunda.

O - Principio de Abierto/Cerrado (Open/Closed Principle - OCP)

El OCP dicta que **las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas a la extensión, pero cerradas a la modificación**. Esto significa que el comportamiento de un módulo puede ser extendido sin necesidad de alterar su código fuente existente. En esencia, una vez que una clase ha sido probada y liberada, no deberíamos tener que modificarla para añadir nuevas funcionalidades.

Ejemplo: Si tienes una clase `CalculadoraArea` que calcula el área de diferentes formas geométricas, en lugar de añadir nuevos `if/else` o `switch` dentro de ella cada vez que aparece una nueva forma (lo que implicaría modificarla), el OCP sugiere que se debería extender su funcionalidad a través de la herencia o la implementación de interfaces. Podrías tener una interfaz `Forma` con un método `calcularArea()`, y luego clases como `Circulo` o `Rectangulo` que implementen esa interfaz. La `CalculadoraArea` entonces operaría sobre la interfaz `Forma`.

L - Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)

El LSP, formulado por Barbara Liskov, afirma que **los objetos de un programa deben ser reemplazables por instancias de sus subtipos sin alterar la corrección de ese programa**. En términos más simples, si tienes una clase `B` que hereda de una clase `A`, entonces `B` debe poder usarse en cualquier lugar donde se espere `A` sin causar problemas. Esto implica que los subtipos no deben cambiar el comportamiento esperado definido por sus supertipos.

Ejemplo: Si tienes una clase `Pájaro` con un método `volar()`, y luego creas una subclase `Pingüino` que también hereda de `Pájaro`, pero los pingüinos no vuelan. Si un código espera un `Pájaro` y llama a `volar()`, y se le pasa un `Pingüino`, el programa podría comportarse de manera inesperada o lanzar un error. El LSP sugiere que `Pingüino` no debería heredar directamente de `Pájaro` si no cumple con el contrato de comportamiento de volar, o que la jerarquía debería rediseñarse (por ejemplo, tener una clase `AnimalQueVuela` y otra `AnimalQueNoVuela`).

I - Principio de Segregación de la Interfaz (Interface Segregation Principle - ISP)

El ISP establece que **los clientes no deberían ser forzados a depender de interfaces que no utilizan**. Es decir, es mejor tener muchas interfaces pequeñas y específicas que una interfaz grande y monolítica. Una interfaz grande podría obligar a las clases a implementar métodos que no necesitan, lo que rompe el Principio de Responsabilidad Única y dificulta el mantenimiento.

Ejemplo: Imagina una interfaz `Trabajador` con métodos `trabajar()`, `comer()`, `dormir()`, `cobrarSueldo()`. Si tienes un `RobotTrabajador` que no necesita `comer()` o `dormir()`, al implementar `Trabajador`, se vería obligado a implementar esos métodos, quizás vacíos o lanzando excepciones. El ISP sugiere dividir `Trabajador` en interfaces más pequeñas como `TrabajadorActivo` (con `trabajar()`), `SerVivo` (con `comer()`, `dormir()`) y `Asalariado` (con `cobrarSueldo()`). Así, `RobotTrabajador` solo implementaría `TrabajadorActivo`.

D - Principio de Inversión de Dependencias (Dependency Inversion Principle - DIP)

El DIP establece que:

1. **Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.**

2. **Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.**

En esencia, este principio promueve el uso de abstracciones (interfaces o clases abstractas) para acoplar las clases, en lugar de depender de implementaciones concretas. Esto reduce el acoplamiento entre los módulos y facilita la flexibilidad y la facilidad de prueba.

Ejemplo: Si una clase `ProcesadorDePedidos` depende directamente de una implementación concreta de `BaseDeDatosSQL` para guardar los pedidos, está fuertemente acoplada a ella. Si se quiere cambiar a una `BaseDeDatosNoSQL`, habría que modificar `ProcesadorDePedidos`. El DIP sugiere que `ProcesadorDePedidos` dependa de una interfaz `RepositorioDePedidos` (una abstracción), y `BaseDeDatosSQL` (o `BaseDeDatosNoSQL`) implementen esa interfaz. Así, `ProcesadorDePedidos` no necesita saber los detalles de la base de datos subyacente.

Aplicar los principios SOLID puede parecer un esfuerzo inicial, pero a largo plazo, resultan en sistemas más robustos, escalables y fáciles de mantener, lo que es crucial en el desarrollo de software moderno.