



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA





UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

ASIGNATURA

PROGRAMACIÓN ORIENTADA A

OBJETOS



Transformamos el mundo desde la Amazonía

Ing. Edwin Gustavo Fernández Sánchez, Mgs.

DOCENTE - PERSONAL ACADÉMICO NO TITULAR OCASIONAL

DIRECTOR DE GESTIÓN DE TECNOLOGÍAS INFORMACIÓN Y COMUNICACIÓN

UNIVERSIDAD ESTATAL AMAZÓNICA





UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

SEMANA

4

DESARROLLO DE LA SEMANA 4: DEL JUE. 02 AL DOM. 05 DE ENERO/2025

Resultado de aprendizaje: Comprender los conceptos fundamentales de la programación orientada a objetos (POO) para el desarrollo de soluciones computacionales de complejidad media.

CONTENIDOS

UNIDAD I: Fundamentos de la programación orientada a objetos

- **Tema 2: Conceptos orientados a objetos**
 - **Subtema 2.2: Características de la programación orientada a objetos**



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Unidad 1

Fundamentos de la programación orientada a objetos

Tema 2

Conceptos orientados a objetos



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Subtema 2.2: Características de la programación orientada a objetos

**PROGRAMACIÓN ORIENTADA A
OBJETOS
(UEA-L-UFB-030)**



Características de la Programación Orientada a Objetos

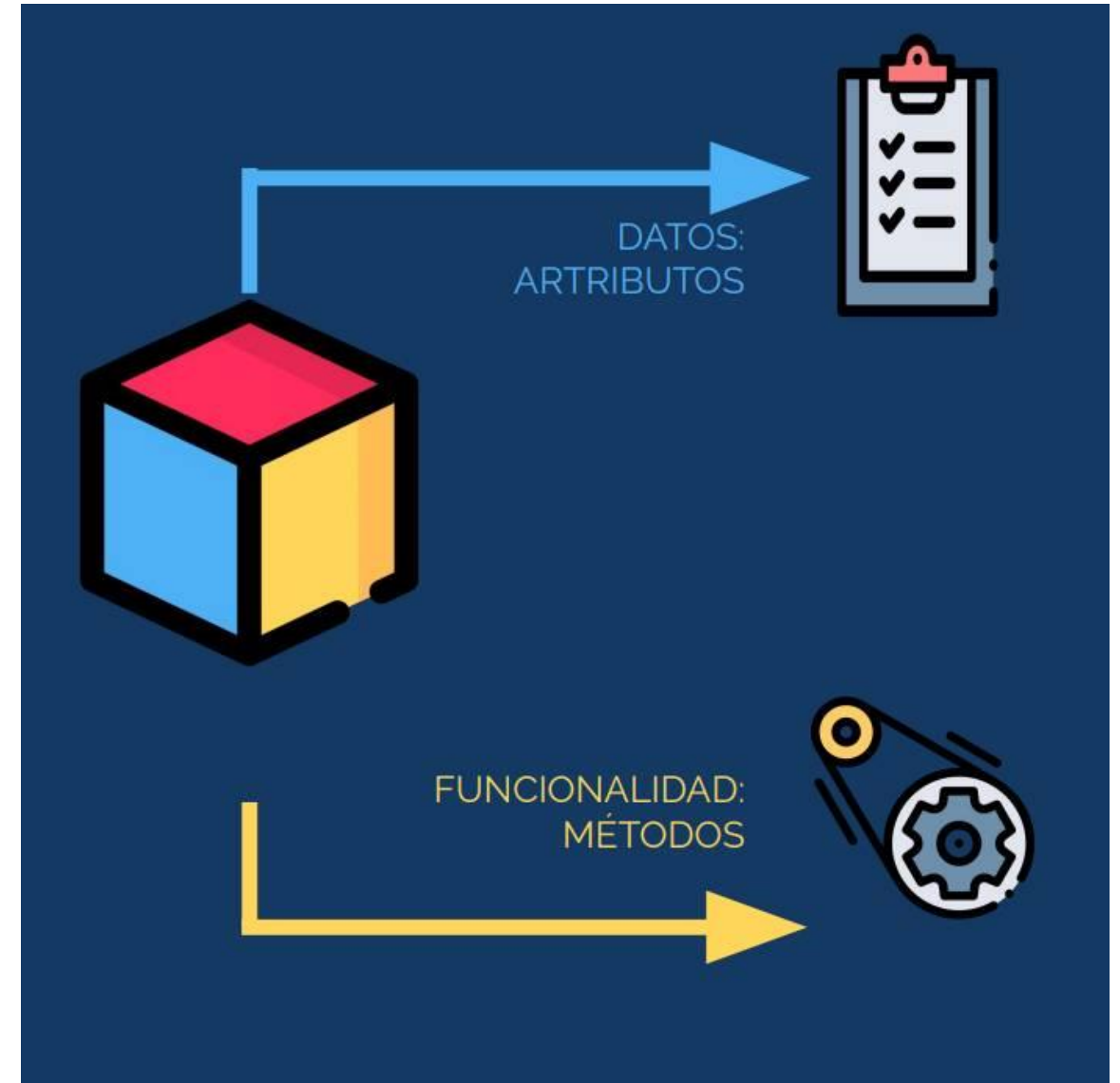
La Programación Orientada a Objetos es un paradigma de programación basado en el concepto de "objetos", que pueden contener datos, en forma de campos (a menudo conocidos como **atributos** o propiedades), y código, en forma de procedimientos (a menudo conocidos como **métodos**).

La POO no es solo una forma de escribir código, sino una manera de **pensar y organizar** el software. Se ha convertido en un estándar crucial en el desarrollo de software moderno debido a su eficacia y eficiencia para modelar sistemas complejos.



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Conceptos fundamentales Programación Orientada a Objetos





UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

POO y el Mundo Real

La Programación Orientada a Objetos (POO) es notablemente eficaz en modelar situaciones y entidades del mundo real.

Esta capacidad es fundamental para entender cómo aplicar la POO en el desarrollo de software.

1. Reflejo del Mundo Real: En la vida real, interactuamos con objetos que tienen propiedades (atributos) y comportamientos (métodos).

De manera similar, en **POO**, modelamos estos objetos utilizando clases y objetos en nuestro código.

Por ejemplo:

Un objeto "Carro" en la vida real tiene propiedades como color, marca y modelo, y comportamientos como acelerar y frenar. En POO, estos se traducen en atributos y métodos de una clase "Carro".



Ejemplo

En este código, la clase **Carro** se define con tres **atributos** (color, marca, modelo) y dos **métodos** (acelerar, frenar).

El método acelerar aumenta la velocidad actual del carro por el incremento proporcionado, mientras que el método frenar la disminuye, asegurándose de que la velocidad no caiga por debajo de 0. Cuando se invocan estos métodos, se imprime una declaración que refleja la acción tomada por el carro.

python

Copy code

```
class Carro:
    def __init__(self, color, marca, modelo):
        self.color = color
        self.marca = marca
        self.modelo = modelo
        self.velocidad = 0

    def acelerar(self, incremento):
        """Aumenta la velocidad del carro."""
        self.velocidad += incremento
        print(f"El {self.marca} {self.modelo} aceleró a {self.velocidad}")

    def frenar(self, decremento):
        """Disminuye la velocidad del carro."""
        self.velocidad = max(0, self.velocidad - decremento)
        print(f"El {self.marca} {self.modelo} frenó a {self.velocidad}")

# Ejemplo de creación y uso de un objeto Carro
mi_carro = Carro('rojo', 'Toyota', 'Corolla')
mi_carro.acelerar(20)
mi_carro.frenar(10)
```



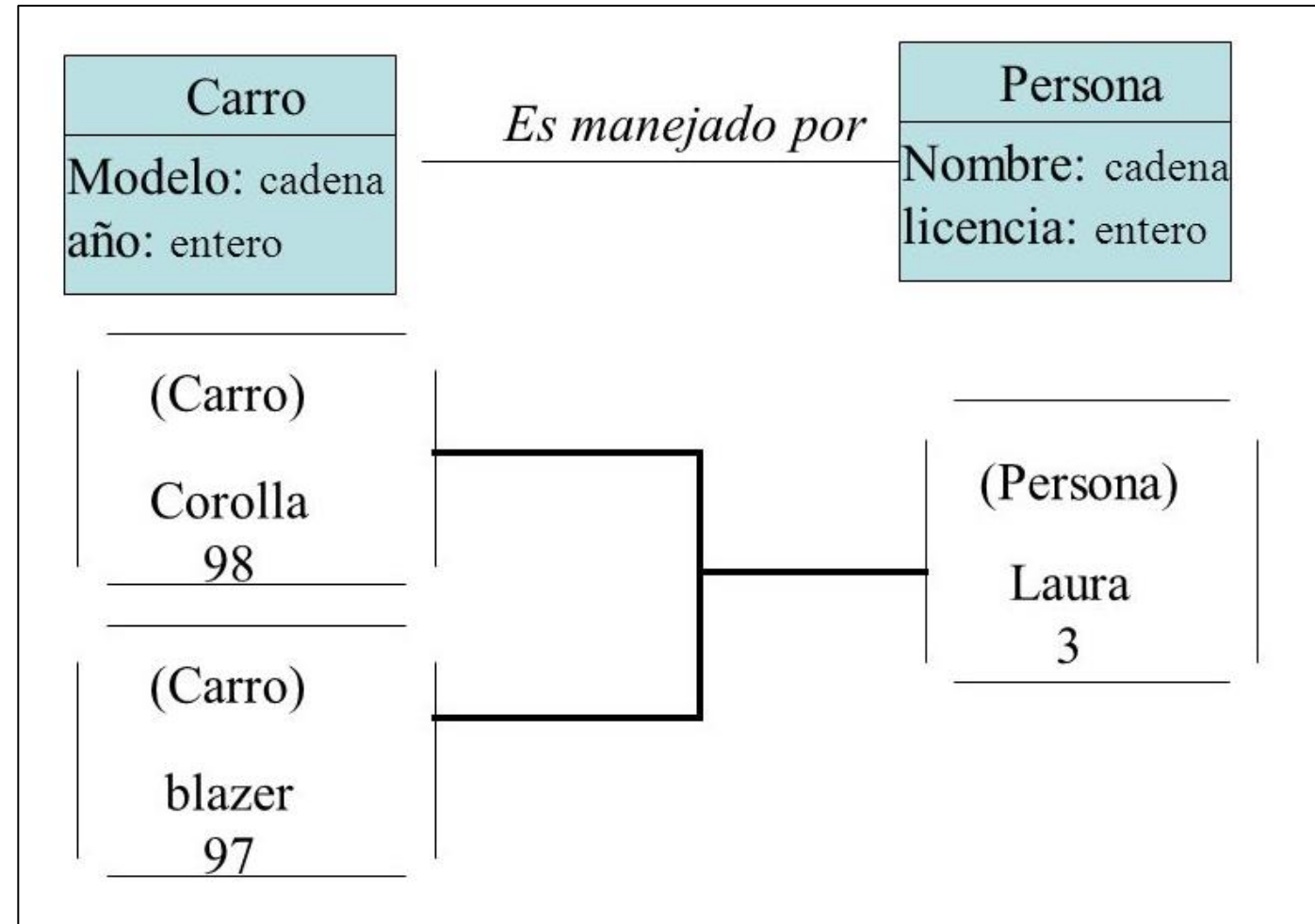


POO y el Mundo Real

2. Relaciones entre Objetos: Así como los objetos en el mundo real interactúan entre sí, en POO, los objetos de software también interactúan.

Por ejemplo:

Un objeto "Persona" puede tener un objeto "Carro". Esta relación se puede modelar en POO donde un objeto "Persona" tiene una referencia a un objeto "Carro".






Ejemplo

En este código, hemos creado dos clases con sus respectivos atributos. La clase **Carro** tiene un método adicional llamado **asignar_conductor**, que establece una relación entre un objeto Carro y un objeto Persona.

También hemos incluido métodos `__str__` en ambas clases para proporcionar una representación en forma de cadena de los objetos cuando se imprimen. El ejemplo final muestra cómo crear instancias de cada clase y cómo relacionarlas.

python

 Copy code

```
class Carro:
    def __init__(self, modelo, anio):
        self.modelo = modelo
        self.anio = anio
        self.conductor = None # Inicialmente, el carro no tiene conductor

    def asignar_conductor(self, persona):
        if isinstance(persona, Persona):
            self.conductor = persona

    def __str__(self):
        return f'Carro {self.modelo} del año {self.anio}, conducido por {self.conductor}'

class Persona:
    def __init__(self, nombre, licencia):
        self.nombre = nombre
        self.licencia = licencia

    def __str__(self):
        return f'Persona {self.nombre} con licencia número {self.licencia}'
```



NOTA (Método `__str__`)

El método `__str__` en Python es un método especial que se define dentro de una clase y se utiliza para devolver una representación de cadena de un objeto.

Cuando utilizas la función `str()` en un objeto o cuando usas la función `print()` para imprimir un objeto, Python automáticamente llama al método `__str__` de ese objeto.

python

Copy code

```
class Carro:
    def __init__(self, color, marca, modelo):
        self.color = color
        self.marca = marca
        self.modelo = modelo
        self.velocidad = 0

    def acelerar(self, incremento):
        """Aumenta la velocidad del carro."""
        self.velocidad += incremento
        print(f"El {self.marca} {self.modelo} aceleró a {self.velocidad}")

    def frenar(self, decremento):
        """Disminuye la velocidad del carro."""
        self.velocidad = max(0, self.velocidad - decremento)
        print(f"El {self.marca} {self.modelo} frenó a {self.velocidad}")

    def __str__(self):
        """Devuelve una representación en cadena del carro."""
        return f"Carro: {self.marca} {self.modelo}, Color: {self.color}"

# Ejemplo de creación y uso de un objeto Carro
mi_carro = Carro('rojo', 'Toyota', 'Corolla')

# Llamada al método __str__
print(mi_carro) # Esto llamará automáticamente al método __str__ y mos
```



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA


Ejemplo Práctico

POO y el Mundo Real

Pensemos en un sistema de gestión de una biblioteca. En la vida real, una biblioteca tiene libros, bibliotecarios, y usuarios. Cada uno de estos puede ser representado como un objeto en nuestro programa.

Un objeto "Libro" podría tener atributos como título, autor y ISBN, y métodos como prestar y devolver. Este enfoque nos permite diseñar un sistema que es intuitivo y cercano a la realidad.

python

 Copy code

```
class Book:
    """Representa un libro en la biblioteca."""

    def __init__(self, title, author, isbn):
        """Inicializa un nuevo libro con título, autor e ISBN."""
        self.title = title
        self.author = author
        self.isbn = isbn
        self.is_borrowed = False

    def borrow(self):
        """Presta el libro si no está actualmente prestado."""
        if not self.is_borrowed:
            self.is_borrowed = True
            return True
        return False

    def return_book(self):
        """Devuelve el libro a la biblioteca."""
        self.is_borrowed = False
```




Modularidad en POO

La modularidad en la Programación Orientada a Objetos (POO) se refiere a la división de un programa en partes más pequeñas, conocidas como módulos. Cada módulo es independiente y contiene todo lo necesario para ejecutar una parte específica de la funcionalidad del programa.





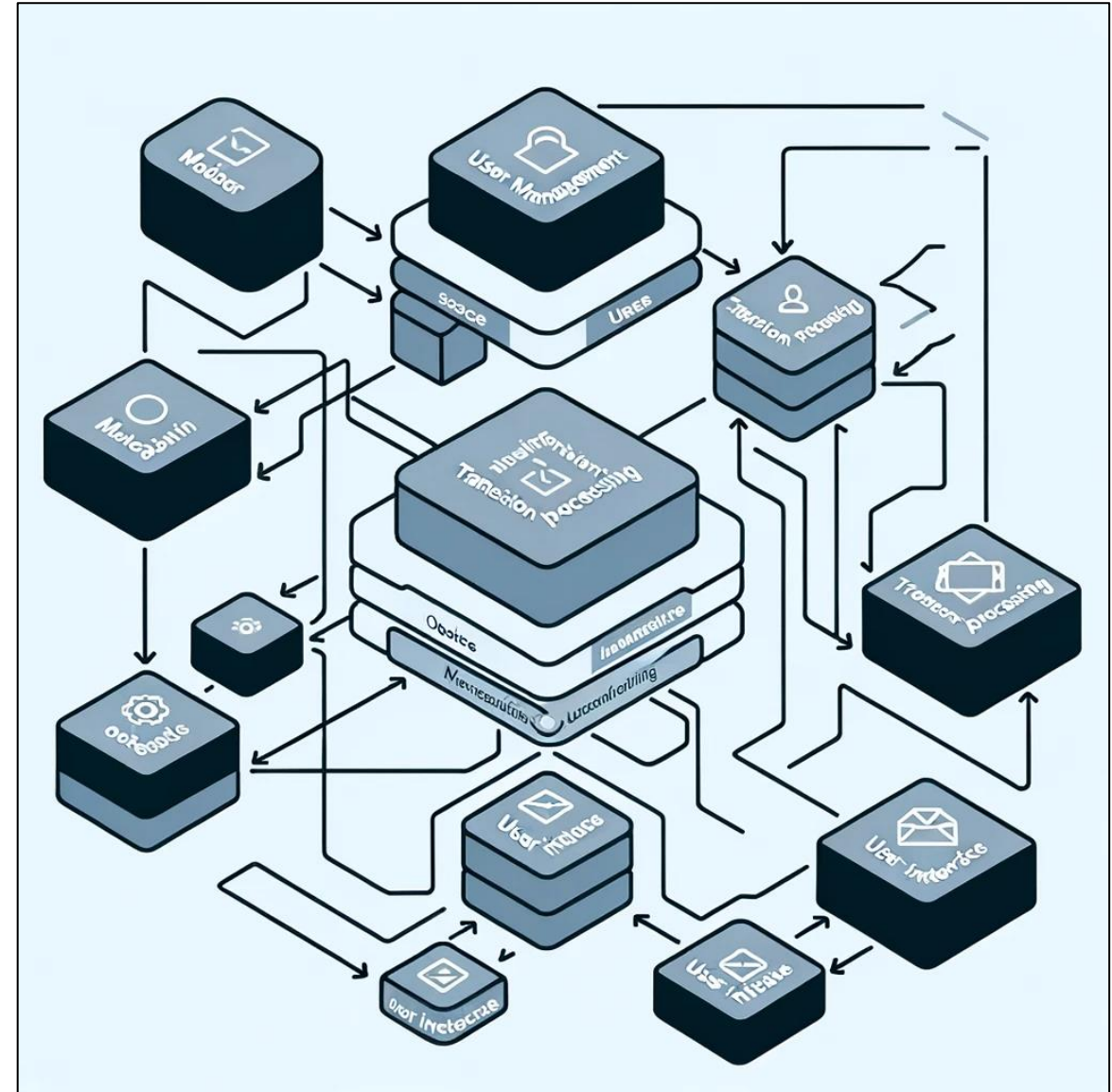
Modularidad en POO

- **Facilita la Mantenibilidad:** Al tener módulos bien definidos, si surge un error en un área específica del código o es necesario realizar una mejora, sólo se debe revisar el módulo correspondiente.
- **Promueve la Reutilización de Código:** Los módulos pueden ser reutilizados en diferentes partes del programa o incluso en proyectos diferentes, lo que ahorra tiempo y esfuerzo.
- **Facilita la Colaboración:** Diferentes equipos o individuos pueden trabajar en módulos separados simultáneamente, lo que mejora la eficiencia del desarrollo de software.
- **Mejora la Comprensión del Código:** Un programa bien modularizado es más fácil de entender ya que cada módulo tiene una responsabilidad claramente definida.

Cómo la POO Apoya la Modularidad

En POO, los módulos se implementan a menudo como clases. Cada clase encapsula los datos y comportamientos relacionados, lo que corresponde a un módulo en el concepto de modularidad.

Ejemplo Práctico: En un sistema de gestión bancaria, podríamos tener un módulo para gestionar las cuentas de los clientes, otro para procesar transacciones y otro para el manejo de la interfaz de usuario. Cada uno de estos módulos sería una clase o un conjunto de clases que trabajan juntas.





UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA