COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

Introducción a los Paradigmas de Programación

¿QUÉ SON?

- Los paradigmas de la programación son estilos que se siguen a la hora de programar un software.
- Diferentes "maneras de hacer las cosas" en la programación.

Tipos

- **Paradigma Imperativo**
  - Programación Estructurada
  - Programación Procedimental
  - Programación Modular
- **Paradigma Declarativo**
  - **Programación Funcional**
  - Programación Lógica
- **Programación Orientada a Objetos**
- **Paradigma Orientado a Componentes**
- **Paradigma Orientado a Eventos**
- Paradigma Reactivo
- Paradigma Paralelo

Paradigma Imperativo

- Describir cómo se deben ejecutar instrucciones paso a paso para cambiar el estado del programa y alcanzar un resultado deseado
- Control del flujo
- Manipulación de variables
- Proyectos: Sistemas Operativos, Navegadores Web…
- Lenguajes: C, C++, Java, Python…

Paradigma Imperativo
Programación Estructurada

- Organización del flujo de control
- Estructuras de control
  - Secuenciales
  - Selección
  - Repetición o bucles

Paradigma Imperativo
Programación Estructurada

- Organización del flujo de control
- Estructuras de control

```
# Calculating the sum of first 10 even numbers
num1 = 2
num2 = 4
num3 = 6
num4 = 8
num5 = 10
num6 = 12
num7 = 14
num8 = 16
num9 = 18
num10 = 20

sum = num1 + num2 + num3 + num4 + num5 + num6 + num7 + num8 + num9 + num10
print("Sum of first 10 even numbers:", sum)
```

Paradigma Imperativo
Programación Estructurada

- Organización del flujo de control
- Estructuras de control

```python
# Calculating the sum of first 10 even numbers using a loop
sum = 0
for i in range(2, 21, 2):
    sum += i

print("Sum of first 10 even numbers:", sum)
```

Paradigma Imperativo
Programación Procedimental

- Organización en procedimientos o funciones
- Uso de variables globales
- Reutilización de código limitada

Paradigma Imperativo
Programación Procedimental

- Organización en procedimientos o funciones
- Uso de variables globales
- Reutilización de código limitada

```python
def main():
    num = 5
    result = 1

    for i in range(1, num + 1):
        result *= i

    print("Factorial of", num, "is", result)

if __name__ == "__main__":
    main()
```

Paradigma Imperativo
Programación Modular

- Módulos independientes
- Cada módulo tiene una funcionalidad específica
- Mayor reutilización y claridad del código

Paradigma Imperativo
Programación Modular

- Módulos independientes
- Cada módulo tiene una funcionalidad específica
- Mayor reutilización y claridad del código

```python
def calculate_factorial(n):
    result = 1

    for i in range(1, n + 1):
        result *= i

    return result

def main():
    num = 5
    fact = calculate_factorial(num)
    print("Factorial of", num, "is", fact)

if __name__ == "__main__":
    main()
```

Paradigma Declarativo

- Describir el resultado deseado, no en cómo lograrlo
- Énfasis en "Qué" en lugar de "Cómo“
- Lógica y Reglas
- Menos control detallado de flujo
- Mayor Nivel de Abstracción
- Proyectos: Bases de datos…
- Lenguajes: SQL, Prolog, Hakell, Python…

Paradigma Declarativo
Programación Funcional

- Trata las operaciones como funciones matemáticas
- Fomenta la inmutabilidad de datos
- Uso de funciones de orden superior
- Énfasis en transformaciones de datos
- Proyectos: Software financiero, Sistemas de inteligencia artificial…
- Lenguajes: Haskell, Lisp, Scala, Python…

Paradigma Declarativo
Programación Funcional

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Approach without functional programming
sum_even_squares = 0
for num in numbers:
    if num % 2 == 0:
        sum_even_squares += num ** 2

print("Sum of squares of even numbers:", sum_even_squares)
```

Paradigma Declarativo
Programación Funcional

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Functional programming approach
even_numbers = filter(lambda x: x % 2 == 0, numbers)
even_squares = map(lambda x: x ** 2, even_numbers)
sum_even_squares = sum(even_squares)

print("Sum of squares of even numbers:", sum_even_squares)
```

Paradigma Declarativo
Programación Lógica

- Se basa en la lógica formal
- Define hechos y reglas
- El ordenador deduce respuestas lógicas
- Ideal para problemas con soluciones basadas en inferencias
- Programas: Procesamiento de Lenguaje Natural (PLN)…
- Lenguajes: Prolog, Mercury, Python (pyke)…

Paradigma Declarativo
Programación Lógica

```
% Knowledge base
parent(juan, ana).
parent(juan, carlos).
parent(ana, maria).

% Logical rule
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

% Queries
?- grandparent(juan, maria).
?- grandparent(juan, carlos).
?- grandparent(ana, maria).
```

Paradigma Declarativo
Programación Lógica

```python
from pyke import knowledge_engine

# Create a knowledge base
engine = knowledge_engine.engine(__file__)

# Define facts in the knowledge base
engine.assert_('family', 'parent(juan, maria)')
engine.assert_('family', 'parent(juan, carlos)')
engine.assert_('family', 'parent(ana, maria)')

# Define logical rule for grandparent
@engine.prove
def grandparent(X, Y):
    return (
        engine.assert_('family', f'parent({X}, {Z})'),
        engine.assert_('family', f'parent({Z}, {Y})')
    )

# Queries
print(grandparent('juan', 'maria'))
print(grandparent('juan', 'carlos'))
print(grandparent('ana', 'maria'))
```

Programación Orientada a Objetos (POO)

- Organiza el código en objetos
- Cada objeto tiene datos y funciones (métodos)
- Encapsulación: oculta los detalles internos
- Herencia: permite crear nuevas clases basadas en otras
- Polimorfismo: objetos de diferentes clases pueden compartir interfaces
- Proyectos: Medios de pago, Videojuegos…
- Lenguajes: Java, C#, C++, Python…

Programación Orientada a Objetos (OOP)

```python
class Animal:
    def __init__(self, name, type):
        self.name = name
        self.type = type

    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Create instances of classes.
my_dog = Dog("Max", "Canine")
my_cat = Cat("Moon", "Feline")

# Call instance methods
print(f"{my_dog.name} does: {my_dog.make_sound()}")
print(f"{my_cat.name} does: {my_cat.make_sound()}")
```

Programación Orientada a Componentes

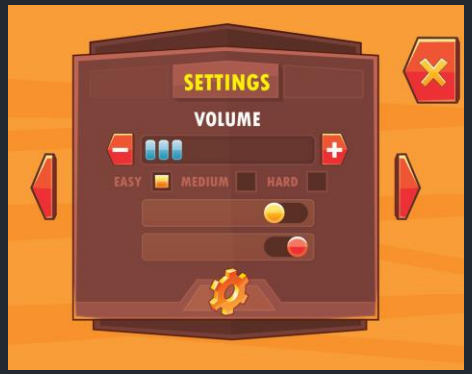
- Diseño basado en componentes independientes
- Reutilización de componentes en múltiples proyectos
- Interconexión de componentes para construir aplicaciones
- Facilita la modularidad y mantenibilidad del código
- Proyectos: Experiencias en Unity…
- Lenguajes: Java, C#, Python…

Programación Orientada a Componentes

```
using UnityEngine;

public class Rotator : MonoBehaviour
{
    public float rotationSpeed = 50.0f;

    private void Start()
    {
        Debug.Log("Rotator component started.");
    }

    private void Update()
    {
        RotateObject();
    }

    private void RotateObject()
    {
        transform.Rotate(Vector3.up * rotationSpeed * Time.deltaTime);
    }
}
```

Programación Orientada a Eventos

- Se basa en la interacción con eventos
- Diseña respuestas a eventos específicos
- Facilita la creación de interfaces interactivas
- Proyectos: Páginas Web, Experiencias en Unity…
- Lenguajes: JavaScript, C#, Python (Tkinter)…



Programación Orientada a Eventos

```python
import tkinter as tk

class EventExample:
    def __init__(self, root):
        self.root = root
        self.root.title("Event Handling Example")

        self.button = tk.Button(self.root, text="Click Me!", command=self.handle_button_click)
        self.button.pack(pady=20)

    def handle_button_click(self):
        print("Button clicked!")

if __name__ == "__main__":
    root = tk.Tk()
    app = EventExample(root)
    root.mainloop()
```

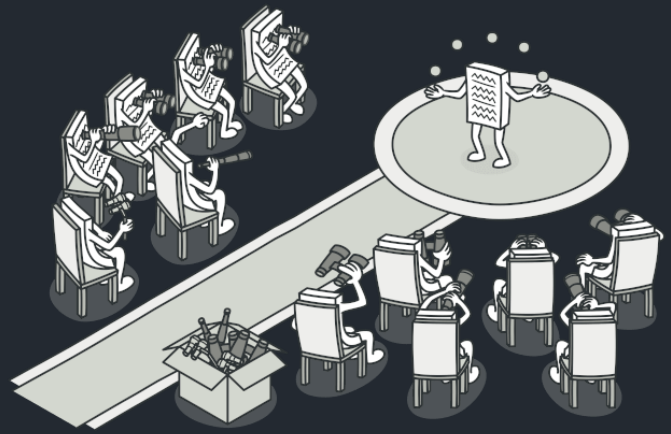
Paradigma Reactivo

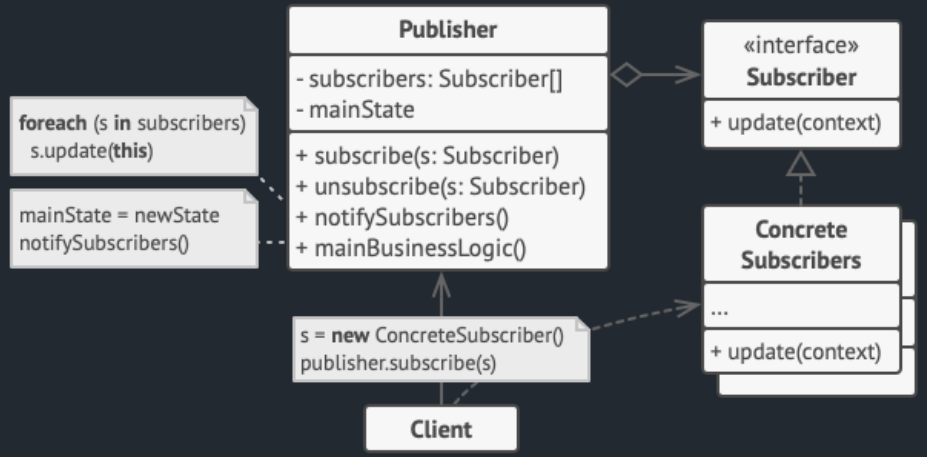
- Reacciona a cambios de datos y eventos
- Actualiza automáticamente las salidas
- Diseño orientado a flujos de datos
- Enfoque en la escalabilidad y la resiliencia
- Proyectos: Plataformas de Streaming de Video, Redes Sociales…
- Lenguajes: JavaScript (React), C#, Java, Python (RxPY)…

NETFLIX

Paradigma Reactivo

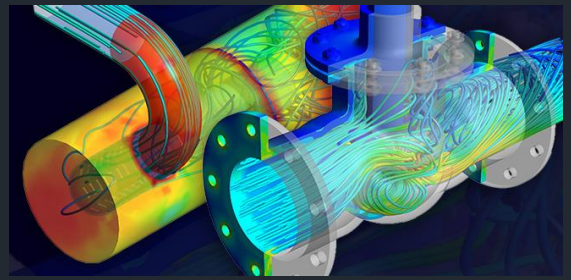
Patrón Observer:





Paradigma Paralelo

- Divide tareas en partes simultáneas
- Aprovecha múltiples procesadores o núcleos
- Mejora la velocidad de ejecución
- Enfoque en la eficiencia y el rendimiento
- Proyectos: Entrenamiento de modelos de IA, Simulación de fluidos…
- Lenguajes: C++, Java, Python…



Paradigma Paralelo

```python
import concurrent.futures

# Function to calculate the square of a number
def calculate_square(number):
    return number * number

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10]

    # Use ThreadPoolExecutor to run tasks in parallel.
    with concurrent.futures.ThreadPoolExecutor() as executor:
        # Map function to numbers and get results.
        results = list(executor.map(calculate_square, numbers))

    print("Results:", results)
```

Enlaces de interés

- Ejemplos: https://github.com/dmorell11/programming-paradigms
- Patrones de diseño: https://refactoring.guru/design-patterns/