```
#include QueAcabeElSemestre.py
```

Strategy

```
{
```

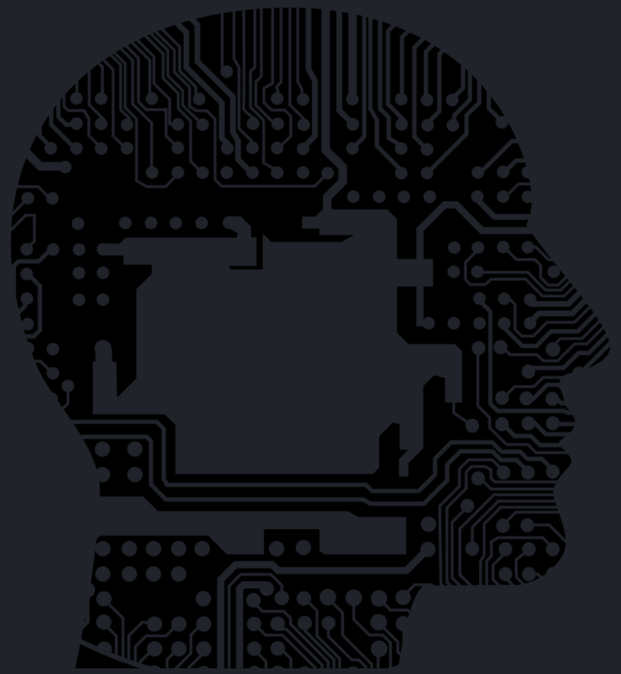
```
    <Es="Patrón de Diseño"/>
```

```
}
```

Contenidos

Definición {

Strategy es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables



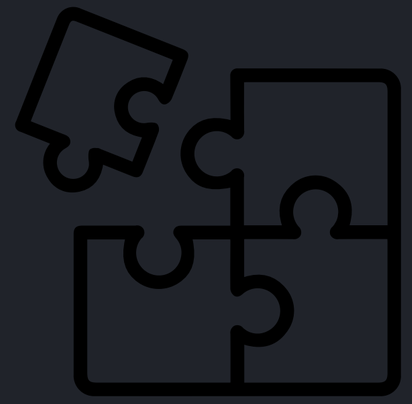
}

Analogía {


VIGOROTH♂ Nv:30
Tas♂ Nv:31
84/ 84
¿Qué debería hacer Tas?
LUCHA
MOCHILA
POKéMON
HUIDA

}

# Representación UML de un strategy pattern
{



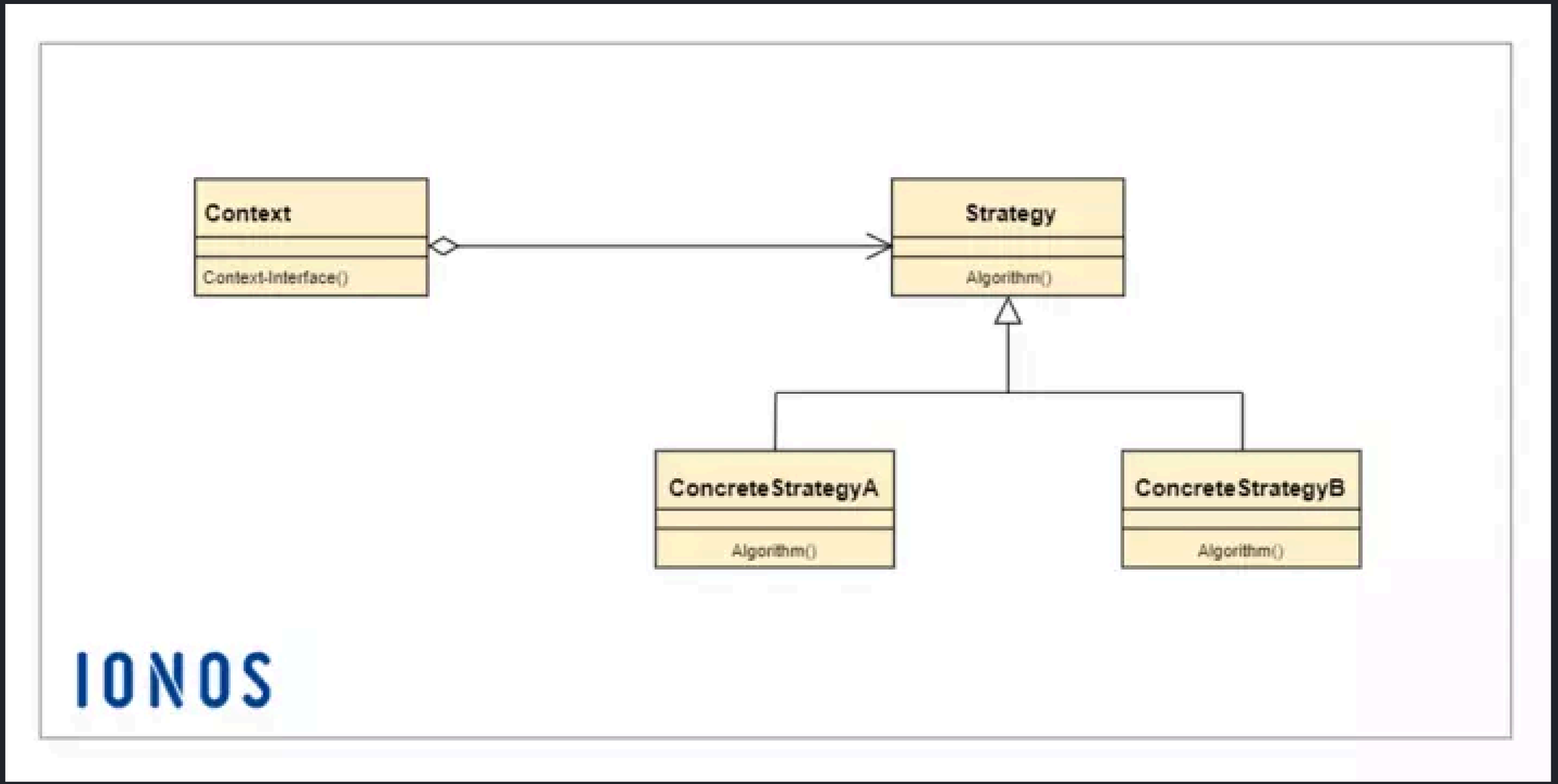
> Sirven para visualizar los patrones de diseño con una notación estandarizada y utilizan caracteres y símbolos especiales.

> El UML establece distintos tipos de diagramas para la programación orientada a objetos. Para representar un strategy pattern, se suelen utilizar los llamados diagramas de clase con al menos tres componentes básicos:

- Context (contexto o clase de contexto)
- Strategy (estrategia o clase de estrategia)
- ConcreteStrategy (estrategia concreta)

}

Estructura {


Context
Context-Interface()
Strategy
Algorithm()
ConcreteStrategyA
Algorithm()
ConcreteStrategyB
Algorithm()
IONOS

}

Aplicabilidad {

- Cuando se quiere utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- Cuando se tenga muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
- Para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
- cuando la clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

}

Funciones mas importantes {

- Está orientado al comportamiento.
- Está orientado a la eficiencia.
- Está orientado al futuro.
- Tiene como objetivo la capacidad de expansión.
- Tiene como objetivo la reutilización.
- Tiene como objetivo optimizar la usabilidad, controlabilidad y configurabilidad del software.
- Requiere consideraciones conceptuales minuciosas.

}

Como implementarlo {

1. En la clase contexto, identifica un algoritmo que tienda a sufrir cambios frecuentes.
2. Declara la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extrae todos los algoritmos y ponlos en sus propias clases. Todas deben implementar la misma interfaz estrategia.
4. En la clase contexto, añade un campo para almacenar una referencia a un objeto de estrategia.
5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

25%

Ganas de dormir presente.

256

Horas invertidas en progra

187.000

Líneas de código inventadas sin saber que hacen

458

Computadores dañados

387

Presentaciones robadas ;)

}

Ventajas {

01

Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.

02

Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.

03

Puedes sustituir la herencia por composición.

04

Principio de abierto/cerrado. Puedes introducir nuevas estrategias sin tener que cambiar el contexto.

}

Desventajas {

01

Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.

02

Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que te permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas.

03

Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.

}

Relación con Otros Patrones

01 Bridge

02 State

03 Adapter

04 Command

05 Decorator

06 Template Method

07 State x2

Algoritmos o Ejemplos {

```python
# Definimos la clase Strategy que será la interfaz común para todos los algoritmos
class Strategy:
    def execute(self, a, b):
        pass

# Implementaciones concretas de Strategy
class Suma(Strategy):
    def execute(self, a, b):
        return a + b

class Resta(Strategy):
    def execute(self, a, b):
        return a - b

class Multiplicacion(Strategy):
    def execute(self, a, b):
        return a * b

class Division(Strategy):
    def execute(self, a, b):
        if b != 0:
            return a / b
        else:
            raise ValueError("No se puede dividir por cero")

# Contexto que utiliza una estrategia concreta
class Contexto:
    def __init__(self, strategy):
        self.strategy = strategy

    def execute_strategy(self, a, b):
        return self.strategy.execute(a, b)

# Uso del patrón
if __name__ == "__main__":
    a = 10
    b = 5

    # Suma
    suma = Contexto(Suma())
    print(f"{a} + {b} = {suma.execute_strategy(a, b)}")

    # Resta
    resta = Contexto(Resta())
    print(f"{a} - {b} = {resta.execute_strategy(a, b)}")

    # Multiplicación
    multiplicacion = Contexto(Multiplicacion())
    print(f"{a} * {b} = {multiplicacion.execute_strategy(a, b)}")

    # División
    division = Contexto(Division())
    print(f"{a} / {b} = {division.execute_strategy(a, b)}")
```

```kotlin
// Interfaz que define el comportamiento de ataque
interface AttackStrategy {
    fun attack()
}

// Clase que implementa la estrategia de ataque aéreo
class AirAttack: AttackStrategy {
    override fun attack() {
        println("Atacando desde el aire")
    }
}

// Clase que implementa la estrategia de ataque terrestre
class GroundAttack: AttackStrategy {
    override fun attack() {
        println("Atacando desde el suelo")
    }
}

// Clase que representa una unidad militar
class MilitaryUnit(var attackStrategy: AttackStrategy) {
    fun attack() {
        attackStrategy.attack()
    }
}

// Creación de una unidad militar con una estrategia de ataque aéreo
val militaryUnit = MilitaryUnit(AirAttack())
militaryUnit.attack() // Imprime "Atacando desde el aire"

// Cambio de la estrategia de ataque a una estrategia terrestre
militaryUnit.attackStrategy = GroundAttack()
militaryUnit.attack() // Imprime "Atacando desde el suelo"
```

}

<!--Progra IV-->

Gracias {

<Por="Steven Grisales y Luis Garzón/>

}