

Singleton

El patrón Singleton es un patrón de diseño creacional que asegura que una clase tenga una única instancia y ofrece un acceso global a ella. Es útil para controlar el acceso a recursos compartidos, como bases de datos o configuraciones.

Abstract Factory

El patrón Abstract Factory es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Este patrón es útil cuando se necesita garantizar que los productos de una familia sean compatibles entre sí.



patrones de diseño

Los patrones de diseño creacionales son técnicas que facilitan la creación de objetos en programación, adaptándose a situaciones específicas. Ayudan a abstraer la instanciación de objetos, promoviendo la flexibilidad y la reutilización del código.

Factory Method

El patrón Factory Method es un patrón de diseño creacional que define una interfaz para crear objetos, pero permite que las subclases decidan qué clase instanciar. Esto promueve la flexibilidad y la extensibilidad en el código.

Builder

El patrón Builder es un patrón de diseño creacional que se utiliza para construir objetos complejos paso a paso. Permite crear diferentes representaciones de un objeto utilizando el mismo proceso de construcción, separando la construcción de la representación.

ejemplos

Factory Method

```
class Product:
    pass

class ConcreteProductA(Product):
    pass

class ConcreteProductB(Product):
    pass

class Creator:
    def factory_method(self):
        pass

class ConcreteCreatorA(Creator):
    def factory_method(self):
        return ConcreteProductA()

class ConcreteCreatorB(Creator):
    def factory_method(self):
        return ConcreteProductB()
```

singleton

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Inicialización adicional
        return cls._instance
```

Builder

```
class Product:
    def __init__(self):
        self.parts = []

    def add(self, part):
        self.parts.append(part)

class Builder:
    def build_part_a(self):
        pass

    def build_part_b(self):
        pass

    def get_result(self):
        pass

class ConcreteBuilder(Builder):
    def __init__(self):
        self.product = Product()

    def build_part_a(self):
        self.product.add("Part A")

    def build_part_b(self):
        self.product.add("Part B")

    def get_result(self):
        return self.product

class Director:
    def __init__(self, builder):
        self.builder = builder

    def construct(self):
        self.builder.build_part_a()
        self.builder.build_part_b()
```

Abstract Factory

```
class ProductA:
    pass

class ProductB:
    pass

class ConcreteProductA1(ProductA):
    pass

class ConcreteProductA2(ProductA):
    pass

class ConcreteProductB1(ProductB):
    pass

class ConcreteProductB2(ProductB):
    pass

class AbstractFactory:
    def create_product_a(self):
        pass

    def create_product_b(self):
        pass

class ConcreteFactory1(AbstractFactory):
    def create_product_a(self):
        return ConcreteProductA1()

    def create_product_b(self):
        return ConcreteProductB1()

class ConcreteFactory2(AbstractFactory):
    def create_product_a(self):
        return ConcreteProductA2()

    def create_product_b(self):
        return ConcreteProductB2()
```