Factory

Created	@June 17, 2023 11:17 PM
tipo	criacional

Factory é um dos Desiner Patterns mais utilizados no desenvolvimento, isso por que ele é um Designer Pattern fácil, de se aplicar, como muito útil para diversos problemas de criação de objetos ou instancia.

Basicamente o que fazemos é criar uma interface, para gerencia a criação de um objeto ou uma instância,

Por exemplo imagine uma loja pet, que apresenta um kit meu primeiro Pet, que tem tudo o que um tutor de primeira viagem precisa, entretanto esse kit e feito especialmente para o tipo de pede que esse tutor tem. Um interface onde o cliente só especifica o animal que tem, e o sistema criar a partir disso isso é um exemplo de utilização do Factory.

Vantagens

- Baixo acoplamento: poisa criação de um objeto pode ser independente da implementação de classe
- O cliente não precisa conhecer o objeto nem as suas nuance-as, e necessário apenas conhecer a interface
- E fácil adicionar outro objetos a fabrica para criar outros tipos de objeto sem o cliente alterar código

É nos temos três tipos de métodos do Factory:

Simple Factory:

Permite que as interfaces criem objetos, sem expor a logica de sua criação. e como seu nome evidência é também o método mais simples de implementação do Factory

```
from abc import ABCMeta, abstractmethod

# principal class - Mother Class
class Animal(metaclass=ABCMeta):

    @abstractmethod
    def emit_sound(self):
        pass

# classes Son
class Dog(Animal):

    def emit_sound(self):
        return "Au Au Au"


class Cat(Animal):
    def emit_sound(self):
        return "Miau Miau"

# factory
class FactoryAnimal:

    def create(self, type_:int):

        """
        Create object animal
```

```
        Args:
            type_(int): type animal create.
            types accept 1 - dog and 0 - cat

        Returns:
            Class Animal:return a object of implemets
            abstract class animal
        """

        if type_ == 0:
            return Cat()
        elif type_ == 1:
            return Dog()
        else:
            raise ValueError(
                "value expect not accept in function")


factory = FactoryAnimal()
factory.create(0).emit_sound()
factory.create(1).emit_sound()
```

```
from abc import ABCMeta, abstractMethod

class Payment(metaclass=ABCMeta):

    @abstractMethod
    def pay(self, amount):
        pass

class CreditCardPayment(Payment):
    def pay(self, amount):
        amount_final = amount + 1.29
        print(f"Realizando pagamento de R${amount_final} com cartão de crédito.")

class BoletoPayment(Payment):
    def pay(self, amount):
        amount_final = amount + 0.5
        print(f"Realizando pagamento de R${amount_final} com boleto bancário.")

class BankTransferPayment(Payment):
    def pay(self, amount):
        amount_final = amount + 0.29
        print(f"Realizando pagamento de R${amount_final} com transferência bancária.")

class PaymentFactory:
    @staticmethod
    def create_payment(payment_type):
        if payment_type == "credito":
            return CreditCardPayment()
        elif payment_type == "boleto":
            return BoletoPayment()
        elif payment_type == "TEF":
            return BankTransferPayment()

# Uso do Simple Factory
payment_type = "credito"
payment = PaymentFactory.create_payment(payment_type)
payment.pay(100.00)
```

E esse padrão é adequado quando

1. O processo de criação de objetos é relativamente simples e pode ser encapsulado em uma única classe, a fábrica.
2. O cliente precisa criar diferentes instâncias de objetos de classes relacionadas, mas não precisa conhecer os detalhes específicos da criação.
3. O cliente pode trabalhar com a interface comum fornecida pelos objetos criados, sem se preocupar com as classes concretas.
4. A lógica de criação de objetos pode ser centralizada e modificada facilmente sem afetar o código do cliente.

Algumas desvantagens:

1. Rigidez na adição de novos tipos de objetos: Podemos gerar complexidade adicional de logica quanto de números de objetos que a fabrica pode criar, sendo que sempre que precisarmos adicionar novos tipos de objetos, é necessário modificar a lógica da fábrica,
2. Acoplamento indireto: Cria um acoplamento indireto entre o cliente e as classes de instanciadas pela fabrica. Alterações na estrutura da classe de fábrica podem exigir modificações no código do cliente.
3. Rigidez da estrutura: oferecendo uma única maneira de criar objetos, isso pode dificultar a criação de objetos, caso precisem de uma lógica complexa ou variações dependendo de certas condições.
4. Violação do princípio da responsabilidade única: A fabrica dependendo da aplicação pode acumular múltiplas reponsabilidades deixando de ser preocupar apenas com a criação do objeto.

Factory Method

Permitem que as interfaces criem objetos, mas adia a decisão para que as subclasses que estão instanciadas dentro do Factory de um grupo, determinem a Classe para a criação do objeto

```
from abc import ABC, abstractmethod

class Product(ABC):
    @abstractmethod
    def operation(self):
        pass

class ConcreteProductA(Product):
    def operation(self):
        print("ConcreteProductA operation")

class ConcreteProductB(Product):
    def operation(self):
        print("ConcreteProductB operation")

class Creator(ABC):
    @abstractmethod
    def factory_method(self):
        pass

    def some_operation(self):
        product = self.factory_method()
        product.operation()

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

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

# Uso do Factory Method
creator_a = ConcreteCreatorA()
creator_a.some_operation()

creator_b = ConcreteCreatorB()
creator_b.some_operation()
```

```
# pruduto -> class abstract - class mother
class Pub(metaclass = ABCMeta):

    @abstractmethod
    def __repr__(self):
        pass

# factory - abstract class - class mother
class Perfil(metaclass = ABCMeta):

    def __init__(self):
        self.publication = []
        self.create_perfil()

    @abstractmethod
    def create_perfil(sefl):
```

```
        pass


    def get_publications(self):
        return self.publication

    def add_publication(self, pub):
        self.publication.append(pub)
        return True

# factory final - classes son
class Facebook(Perfil):

    def create_perfil(self):
        self.add_publication(Feed())
        self.add_publication(Story())


class Instagram(Perfil):

    def create_perfil(self):
        self.add_publication(Feed())
        self.add_publication(Story())
        self.add_publication(Carousel())

# final product -> classes son
class Carousel(Pub):

    def __repr__(self):
        return "Carousel"


class Story(Pub):

    def __repr__(self):
        return "Story"

class Feed(Pub):

    def __repr__(self):
        return "Feed"


print(Facebook().get_publications())
print(Instagram().get_publications())
print(Instagram().get_publications()[0].__str__())
```

Adequado

1. Criação de objetos complexos: Quando a criação de um objeto envolve uma lógica complexa, o Method Factory pode ser usado para encapsular essa lógica em um método dedicado. Isso simplifica o código do cliente e facilita a compreensão da lógica de criação.
2. Flexibilidade na criação de objetos: Se você precisa permitir diferentes maneiras de criar objetos ou precisa lidar com múltiplas variantes ou famílias de objetos, o Method Factory pode ser aplicado. Ele fornece uma maneira flexível de criar objetos, permitindo a introdução de novos métodos de criação ou subclasses sem modificar o código existente.
3. Aplicação de princípios de design: como o princípio aberto/fechado (Open/Closed Principle), que promove a extensibilidade sem modificar o código existente, e o princípio da inversão de dependência (Dependency Inversion Principle), que desacopla o código do cliente da implementação concreta.

Vantagens

1. Encapsulamento: criação de objetos em uma classe separada, isolando a lógica de criação em um único local. Isso permite que o código do cliente fique desacoplado dos detalhes implementação.
2. Abstração: promove o uso de interfaces ou classes abstratas para definir o método de criação, tornando o código mais flexível e extensível.
3. Reutilização de código: A lógica de criação pode ser compartilhada entre várias classes, evitando a duplicação e promovendo a modularidade do código. (testes também podem ser reutilizados).
4. Separação de responsabilidades: Cada classe se mantem á uma única responsabilidade, seguindo o princípio da responsabilidade única (Single Responsibility Principle).
5. Flexibilidade: Para Substituir ou estender a lógica de criação de objetos. Assim facilitando o adição de novos objetos, assim respeitando o principio de aberto e fechado.

6. Acoplamento reduzido: Cria uma interface que permite que o cliente fique refém apenas da interface para instanciação do objetos.

Desvantagem

1. Complexidade adicional e Abstração: Pode adicionar complexidade ao código, especialmente quando há muitas variantes ou famílias de objetos. Onde a hierarquia pode ser um problema para ser mantida. Além da camada adicional de abstração que em alguns casos pode ser mais simples instanciar o objeto diretamente.
2. Overhead de criação: Introduzir um overhead adicional no processo de criação de objetos. Em vez de simplesmente instanciar uma classe diretamente, é necessário chamar o método de fábrica correspondente, o que pode aumentar a quantidade de código necessário para criar objetos.
3. Rigidez da estrutura: Para objetos que precisam de uma criação mais flexivel isso pode ser complicado de introduzir, assim dificultando a adição de novas objetos.
4. Acoplamento indireto: Cria um acoplamento indireto entre o cliente e as classes de instanciadas pela fabrica. Alterações na estrutura da classe de fábrica podem exigir modificações no código do cliente.
5. Dificuldade de teste de unidade: O teste de unidades que envolve classes que dependem sendo necessário a criação de mocks ou stubs personalizados.

Abstract Factory:

Uma interface para criar objetos que tem uma determinada relação (considerados famílias muitas vezes, ou dependentes), entretanto não expõem suas classes o padrão fornece objetos de outra fabrica que internamente, cria outros objetos,

O padrão Abstract Factory é útil quando você precisa criar conjuntos de objetos relacionados que compartilham uma mesma temática ou estão interconectados. Ele fornece uma maneira de criar objetos relacionados sem depender de suas classes concretas, permitindo que você altere as famílias de objetos a serem criadas sem modificar o código do cliente.

A estrutura básica do padrão Abstract Factory envolve as seguintes entidades:

1. Abstract Factory: Define a interface para as fábricas concretas. Ela contém métodos para criar os objetos relacionados.
2. Concrete Factory: Implementa a interface da Abstract Factory para criar objetos concretos de uma família específica. Cada Concrete Factory é responsável por criar objetos de uma família de produtos relacionados.
3. Abstract Product: Define a interface para os produtos concretos que serão criados pelas fábricas concretas.
4. Concrete Product: Implementa a interface do Abstract Product para criar objetos concretos.
5. Client: Utiliza a Abstract Factory e os produtos abstratos para criar objetos relacionados, sem conhecer as classes concretas envolvidas.

```
from abc import ABC, abstractmethod

# Interface para os produtos A
class AbstractProductA(ABC):
    @abstractmethod
    def operation_a(self) -> str:
        pass

# Implementação concreta do produto A para uma variante específica
class ConcreteProductA1(AbstractProductA):
    def operation_a(self) -> str:
        return "Operação A1"

# Implementação concreta do produto A para outra variante específica
class ConcreteProductA2(AbstractProductA):
    def operation_a(self) -> str:
        return "Operação A2"

# Interface para os produtos B
class AbstractProductB(ABC):
    @abstractmethod
```

```
    def operation_b(self) -> str:
        pass

# Implementação concreta do produto B para uma variante específica
class ConcreteProductB1(AbstractProductB):
    def operation_b(self) -> str:
        return "Operação B1"

# Implementação concreta do produto B para outra variante específica
class ConcreteProductB2(AbstractProductB):
    def operation_b(self) -> str:
        return "Operação B2"

# Interface da fábrica abstrata
class AbstractFactory(ABC):
    @abstractmethod
    def create_product_a(self) -> AbstractProductA:
        pass

    @abstractmethod
    def create_product_b(self) -> AbstractProductB:
        pass

# Implementação concreta da fábrica para uma variante específica
class ConcreteFactory1(AbstractFactory):
    def create_product_a(self) -> AbstractProductA:
        return ConcreteProductA1()

    def create_product_b(self) -> AbstractProductB:
        return ConcreteProductB1()

# Implementação concreta da fábrica para outra variante específica
class ConcreteFactory2(AbstractFactory):
    def create_product_a(self) -> AbstractProductA:
        return ConcreteProductA2()

    def create_product_b(self) -> AbstractProductB:
        return ConcreteProductB2()

# Cliente
def client_code(factory: AbstractFactory) -> None:
    product_a = factory.create_product_a()
    product_b = factory.create_product_b()

    print(product_a.operation_a())
    print(product_b.operation_b())

# Exemplo de uso
print("Cliente usando ConcreteFactory1:")
client_code(ConcreteFactory1())

print("Cliente usando ConcreteFactory2:")
client_code(ConcreteFactory2())
```

Adequado

1. Tem similiar as mesma vantagens que o Method Factory
2. Garantia de consistência: O padrão Abstract Factory assegura que todos os objetos criados pela fábrica são consistentes entre si. A fábrica é responsável por criar objetos que se complementam e seguem uma mesma temática ou estilo. Isso ajuda a evitar incompatibilidades e inconsistências entre objetos relacionados.

Vantagens

São as mesma do Method Factory

Desvantagens

1. Aumento da complexidade e da Abstração? temos mais classes para gerenciar e a logica de implementação da criação de um método aumenta
2. Grande quantidade de Classes: A mais classes para gerenciar o que a longo prazo pode ser um problema para manter e administrar, pois existe muitos passos até chagar na fabrica concreta

3. Dificuldade no Gerenciamento da logica? A mediada que as famílias crescem fica mais difícil, conseguir gerenciar, e muitas vezes e fácil se perder em meio a logica.