МИНЕСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧЕРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«Брестский государственный технический университет»

Кафедра «Интеллектуальные информационные технологии»

Лабораторная работа №4

По дисциплине «Объектно-ориентированное программирование»

За 5 семестр

Тема: «Паттерны проектирование»

Выполнила:

студентка 3 курса

группы АС-56

Карпенко М.В.

Проверил:

Михняев А.Л.

Брест 2021

Цель: реализовать два любых паттерна проектирование

Код программы:

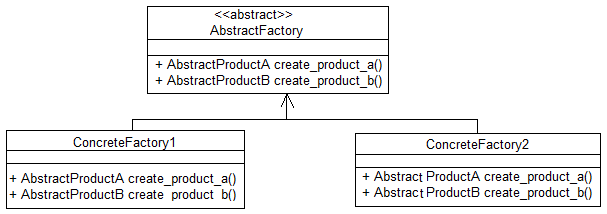
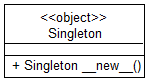
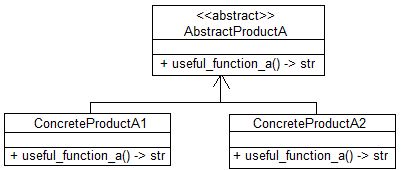
from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
  
  
class AbstractFactory(ABC):  
 @abstractmethod  
 def create\_product\_a(self) -> AbstractProductA:  
 pass  
  
 @abstractmethod  
 def create\_product\_b(self) -> AbstractProductB:  
 pass  
  
  
class ConcreteFactory1(AbstractFactory):  
 def create\_product\_a(self) -> AbstractProductA:  
 return ConcreteProductA1()  
  
 def create\_product\_b(self) -> AbstractProductB:  
 return ConcreteProductB1()  
  
  
class ConcreteFactory2(AbstractFactory):  
 def create\_product\_a(self) -> AbstractProductA:  
 return ConcreteProductA2()  
  
 def create\_product\_b(self) -> AbstractProductB:  
 return ConcreteProductB2()  
  
  
class AbstractProductA(ABC):  
 @abstractmethod  
 def useful\_function\_a(self) -> str:  
 pass  
  
class ConcreteProductA1(AbstractProductA):  
 def useful\_function\_a(self) -> str:  
 return "The result of the product A1."  
  
  
class ConcreteProductA2(AbstractProductA):  
 def useful\_function\_a(self) -> str:  
 return "The result of the product A2."  
  
  
class AbstractProductB(ABC):  
 @abstractmethod  
 def useful\_function\_b(self) -> None:  
 pass  
  
 @abstractmethod  
 def another\_useful\_function\_b(self, collaborator: AbstractProductA) -> None:  
 pass  
  
class ConcreteProductB1(AbstractProductB):  
 def useful\_function\_b(self) -> str:  
 return "The result of the product B1."  
  
  
 def another\_useful\_function\_b(self, collaborator: AbstractProductA) -> str:  
 result = collaborator.useful\_function\_a()  
 return f"The result of the B1 collaborating with the ({result})"  
  
  
class ConcreteProductB2(AbstractProductB):  
 def useful\_function\_b(self) -> str:  
 return "The result of the product B2."  
  
 def another\_useful\_function\_b(self, collaborator: AbstractProductA) -> str:  
  
 result = collaborator.useful\_function\_a()  
 return f"The result of the B2 collaborating with the ({result})"  
  
  
def client\_code(factory: AbstractFactory) -> None:  
  
 product\_a = factory.create\_product\_a()  
 product\_b = factory.create\_product\_b()  
  
 print(f"{product\_b.useful\_function\_b()}")  
 print(f"{product\_b.another\_useful\_function\_b(product\_a)}", end="")  
  
class Singleton(object):  
 def \_\_new\_\_(cls):  
 if not hasattr(cls, 'instance'):  
 cls.instance = super(Singleton, cls).\_\_new\_\_(cls)  
 return cls.instance  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 s = Singleton()  
 print("Object created", s)  
 s1 = Singleton()  
 print("Object created", s1)  
  
 print("Client: Testing client code with the first factory type:")  
 client\_code(ConcreteFactory1())  
  
 print("\n")  
  
 print("Client: Testing the same client code with the second factory type:")  
 client\_code(ConcreteFactory2())

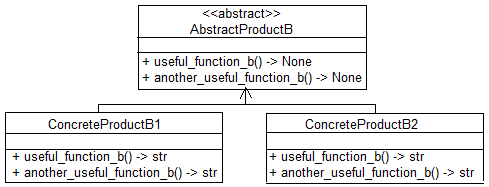
Результат работы программы:

Object created <\_\_main\_\_.Singleton object at 0x000001C860EF5F70>  
Object created <\_\_main\_\_.Singleton object at 0x000001C860EF5F70>  
Client: Testing client code with the first factory type:  
The result of the product B1.  
The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type:  
The result of the product B2.  
The result of the B2 collaborating with the (The result of the product A2.)  
Process finished with exit code 0

UML-диаграмма:



Вывод: реализовала два паттерна: Одиночка и Абстрактная фабрика