Usman Institute of Technology

Department of Computer Science

Course Code: SE308

Course Title: Software Design and Architecture

Fall 2022

Lab 06

OBJECTIVE: Working on Design Patterns

- To Understand Creational Design Patterns.
- To implement Single, Factory and Abstract Factory Design Patterns

Student Information

Student Name	
Student ID	
Date	

Assessment

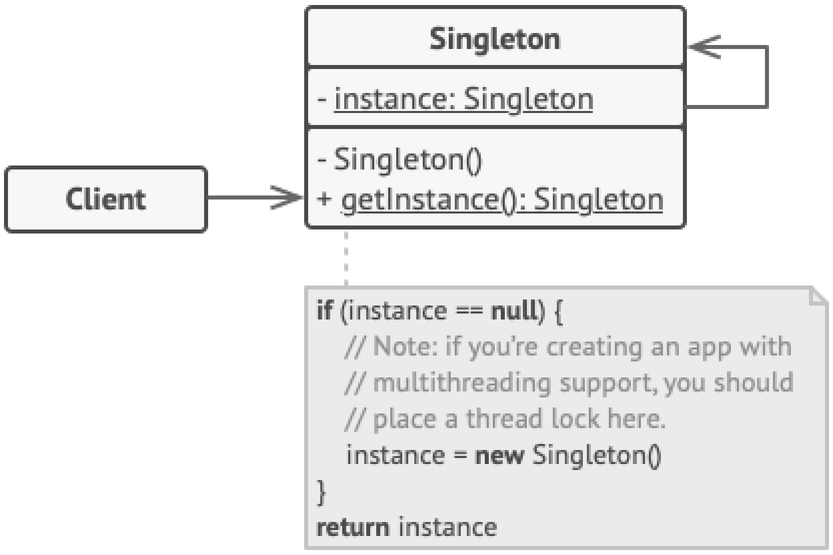
Marks Obtained	
Remarks	
Signature	

Usman Institute of Technology
Department of Computer Science
SE308 - Software Design and Architecture
Lab 06

Singleton Design Pattern

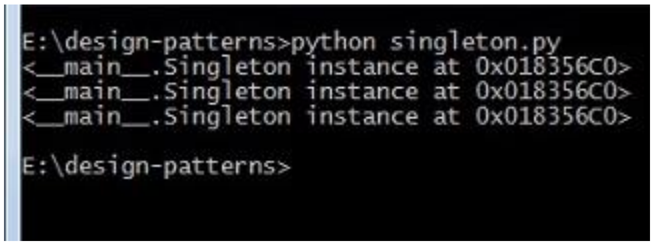
Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

UML class diagram


Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
if (instance == null) {
// Note: if you're creating an app with
// multithreading support, you should
// place a thread lock here.
instance = new Singleton()
}
return instance

```
class Singleton:
  __instance = None
  @staticmethod
  def getInstance():
    """ Static access method. """
    if Singleton.__instance == None:
      Singleton()
    return Singleton.__instance
  def __init__(self):
    """ Virtually private constructor. """
    if Singleton.__instance != None:
      raise Exception("This class is a singleton!")
    else:
      Singleton.__instance = self
s = Singleton()
print s

s = Singleton.getInstance()
print s

s = Singleton.getInstance()
print s
```

Output

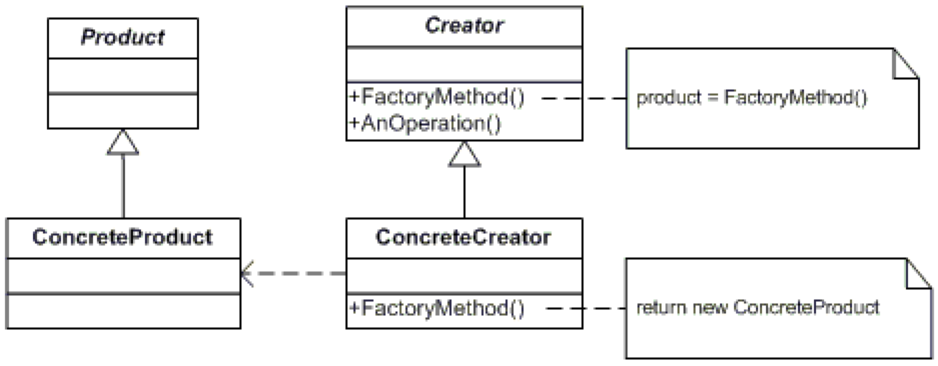


```
E:\design-patterns>python singleton.py
<__main__.Singleton instance at 0x018356C0>
<__main__.Singleton instance at 0x018356C0>
<__main__.Singleton instance at 0x018356C0>

E:\design-patterns>
```

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

UML class diagram



Class Diagram of Factory Method

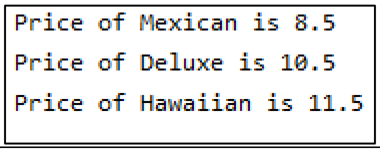
Participants

The classes and objects participating in this pattern are:

- **Product**
  - defines the interface of objects the factory method creates
- **ConcreteProduct**
  - implements the Product interface
- **Creator**
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- **ConcreteCreator**
  - overrides the factory method to return an instance of a ConcreteProduct.

Example in Python

```python
class Pizza(object):
    def __init__(self):
        self._price = None

    def get_price(self):
        return self._price

class MexicanPizza(Pizza):
    def __init__(self):
        self._price = 8.5

class DeluxePizza(Pizza):
    def __init__(self):
        self._price = 10.5

class HawaiianPizza(Pizza):
    def __init__(self):
        self._price = 11.5

class PizzaFactory(object):
    @staticmethod
    def create_pizza(pizza_type):
        if pizza_type == 'Mexican':
            return MexicanPizza()
        elif pizza_type == 'Deluxe':
            return DeluxePizza()
        elif pizza_type == 'Hawaiian':
            return HawaiianPizza()

if __name__ == '__main__':
    for pizza_type in ('Mexican', 'Deluxe', 'Hawaiian'):
          print('Price of {0} is {1}'.format(pizza_type,
PizzaFactory.create_pizza(pizza_type).get_price()))
```



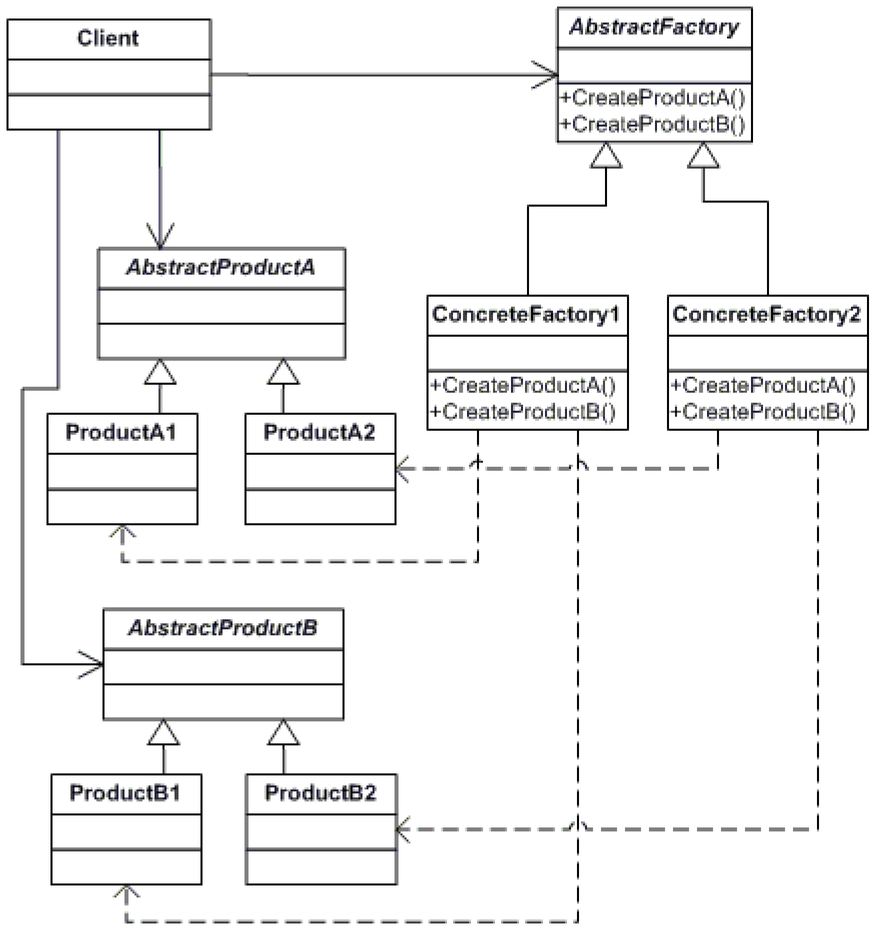
```
Price of Mexican is 8.5
Price of Deluxe is 10.5
Price of Hawaiian is 11.5
```

Abstract Factory

Definition

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **AbstractFactory**
  - declares an interface for operations that create abstract products
- **ConcreteFactory**
  - implements the operations to create concrete product objects
- **AbstractProduct**
  - declares an interface for a type of product object
- **Product**
  - defines a product object to be created by the corresponding concrete factory
  - implements the AbstractProduct interface
- **Client**
  - uses interfaces declared by AbstractFactory and AbstractProduct classes

Example in Python

```python
class Door:
    def getDescription(self):
        pass
class WoodenDoor(Door):
    def getDescription(self):
        print ('I am a wooden door')
class IronDoor(Door):
    def getDescription(self):
        print ('I am an iron door')

class DoorFittingExpert:
    def getDescription(self):
        pass

class Welder(DoorFittingExpert):
    def getDescription(self):
        print ('I can only fit iron doors')

class Carpenter(DoorFittingExpert):
    def getDescription(self):
        print ('I can only fit wooden doors')

class DoorFactory:
    def makeDoor(self):
        pass

    def makeFittingExpert(self):
        pass

class WoodenDoorFactory(DoorFactory):
    def makeDoor(self):
        return WoodenDoor()

    def makeFittingExpert(self):
        return Carpenter()

class IronDoorFactory(DoorFactory):
    def makeDoor(self):
        return IronDoor()

    def makeFittingExpert(self):
        return Welder()

if __name__ == '__main__':
    woodenFactory = WoodenDoorFactory()

    door = woodenFactory.makeDoor()
    expert = woodenFactory.makeFittingExpert()

    door.getDescription()
    expert.getDescription()

    ironFactory = IronDoorFactory()

    door = ironFactory.makeDoor()
    expert = ironFactory.makeFittingExpert()

    door.getDescription()
    expert.getDescription()
```

```
I am a wooden door
I can only fit wooden doors
I am an iron door
I can only fit iron doors
```

Student Tasks:

Class Task

1. For Factory Pattern, Abstract Factory Pattern

   a. Generate (from StarUML) UML diagram of the above patterns
      Tools - > Apply Pattern - > Pattern Repository -> GoF
   a. Compare your generated UML diagram with given code (example in python)
   b. Convert your generated UML diagram according to the given code
   c. Run the code and analyze the output

Home Task

Think about a real life example of the above implemented design patterns, and try to implement in python programming language