Lab 4 Creational Design Pattern

Factory Design Pattern

We will discuss one example of factory design pattern just to give you an overview of the example.

- We have two kinds of products that are being developed by the machines, let's say product A and product B and both of these products provide a method work.
- And along with that, we have two machines, machine A, machine B, they also provide the method work and similar to the product, they are also derived from an abstract class machine, which provides an abstract method for work.

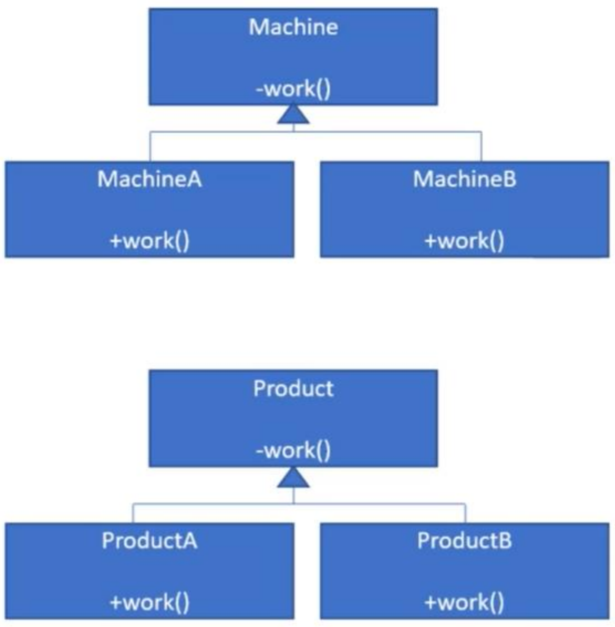


Fig 1 Product and Machines Abstract Classes

- So, we have the products and the machines here and for creating these products and machines, we have the product factory and machine factory, and both of these factories provide the method, create object and using this create object, they create the instances of product A and product B to build a machine A and machine B.
- These two factories, they are also derived from the abstract factory, we have a strict factory which has the abstract method create object, and the implementation or the concrete

implementation of this create object method is provided by these two factories, machine factory and product factory.

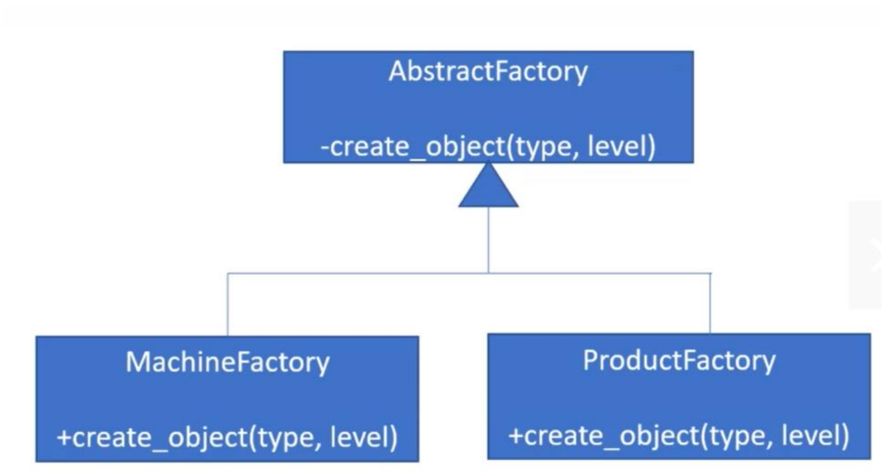
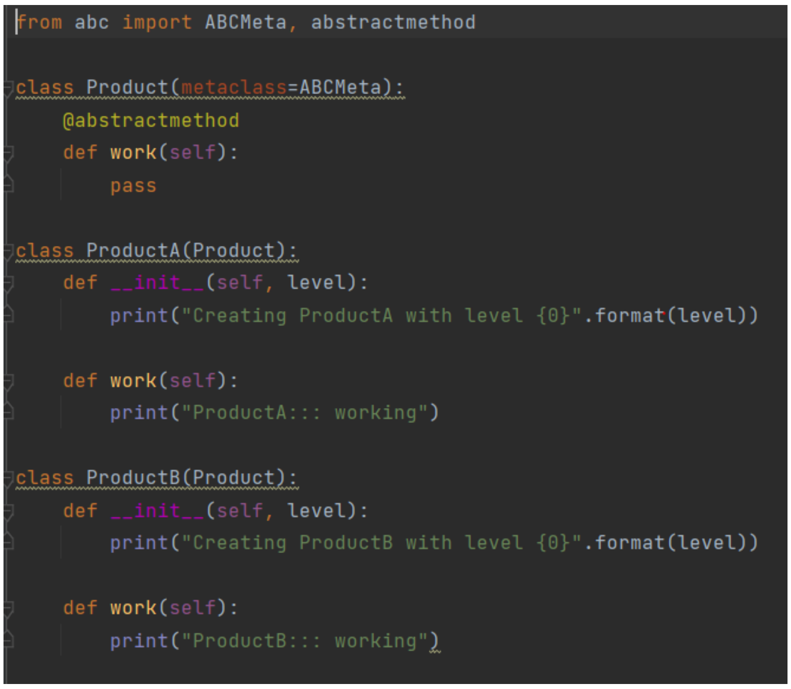

AbstractFactory
-create_object(type, level)
MachineFactory
+create_object(type, level)
ProductFactory
+create_object(type, level)

Fig 2 Abstract Factory

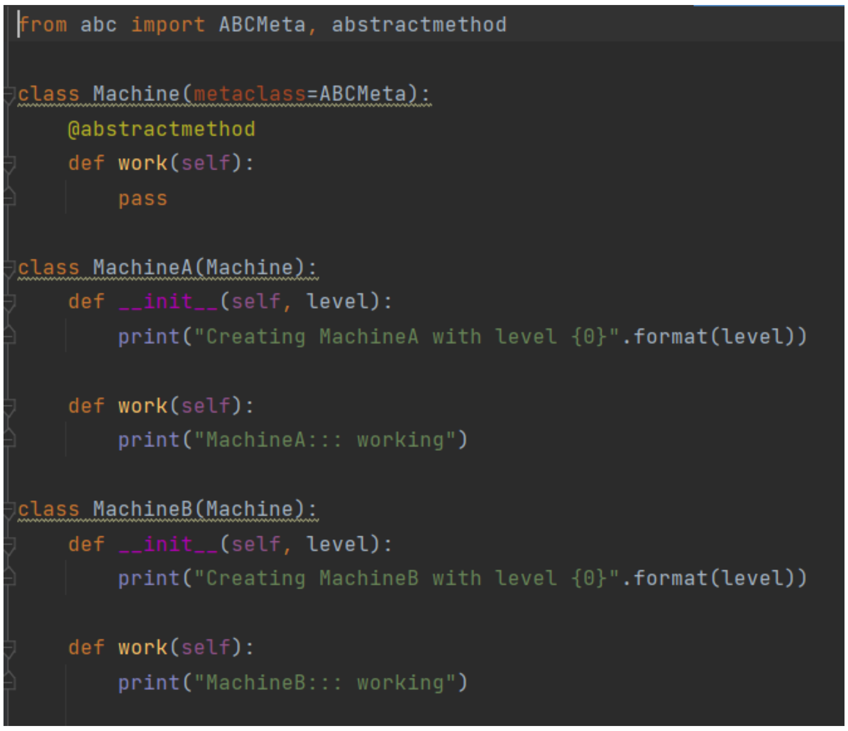
- In these methods, they create the instances of these classes, machine A and machine B or product A and Product B. So, this is the overall view of the example which we are going to discuss in this lab.
- First of all, let's say we have this class product, which is an abstract class and, in this class, we have the abstract method of work.
- We did our two classes, product A and product B, and in these classes, product A and product B, we define the method of work. And similar definition is there in product class as well.



```
from abc import ABCMeta, abstractmethod

class Product(metaclass=ABCMeta):
    @abstractmethod
    def work(self):
        pass

class ProductA(Product):
    def __init__(self, level):
        print("Creating ProductA with level {0}".format(level))

    def work(self):
        print("ProductA::: working")

class ProductB(Product):
    def __init__(self, level):
        print("Creating ProductB with level {0}".format(level))

    def work(self):
        print("ProductB::: working")
```

Fig 3 Product Class

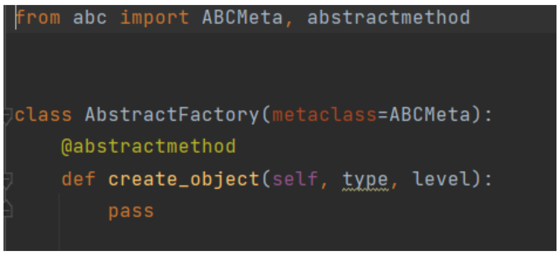
- We have this class product, which is an abstract class, and from this class we are driving product A and product B containing method work.
- Similar to this product, we have the machines, we have an abstract class machine which defines the method work. It is an abstract method. Implementation of this method goes into the desired classes, machine A and machine B. Machine A and machine B, they are both derived from the class machine. So, they have to provide the implementation of method work.



```python
from abc import ABCMeta, abstractmethod

class Machine(metaclass=ABCMeta):
    @abstractmethod
    def work(self):
        pass

class MachineA(Machine):
    def __init__(self, level):
        print("Creating MachineA with level {0}".format(level))

    def work(self):
        print("MachineA::: working")

class MachineB(Machine):
    def __init__(self, level):
        print("Creating MachineB with level {0}".format(level))

    def work(self):
        print("MachineB::: working")
```

Fig 4 Machine Class

- Coming to the factory site, we are having two factories, machine factory and product factory, and they both get derived from the abstract factory. So first of all, let's see the abstract factory.
- So, we have an abstract factory class, which is an abstract class, and it contains an abstract method and Create object. From this abstract factory class, we derived a machine factory.

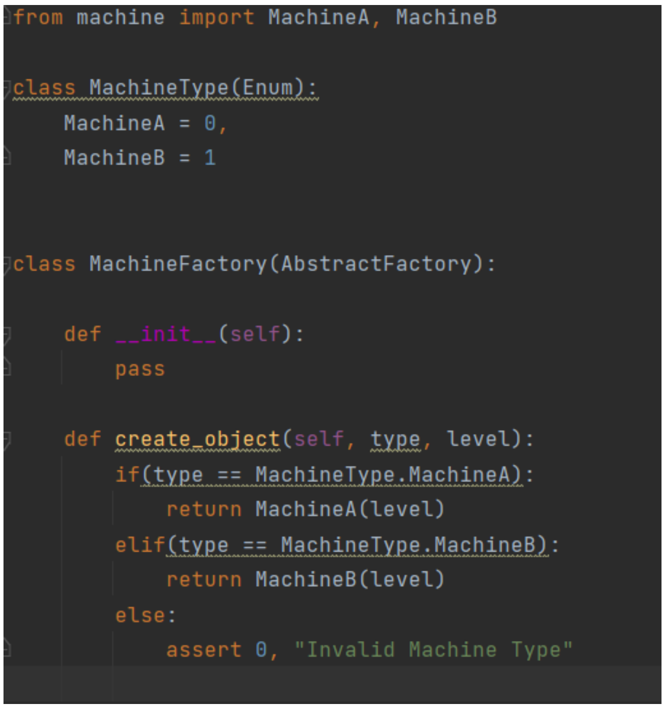


```python
from abc import ABCMeta, abstractmethod


class AbstractFactory(metaclass=ABCMeta):
    @abstractmethod
    def create_object(self, type, level):
        pass
```

Fig 5 Abstract Factory

Here is the machine factory and it is being derived from abstract factory. It needs to provide the method to create object because create object method is abstract in the abstract factory class. It accepts two arguments type and level, and depending upon the type, it creates either of machine A or machine B.

With the given level, right, so this is, again, a simple matter for machine factory and similar to this machine factory, we have a product factory again, it receives two arguments type and level. And depending upon the type, it creates either product, a product to be with the given level.

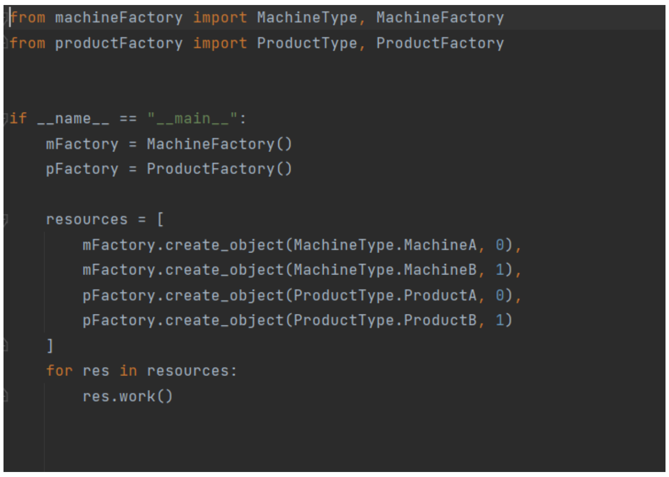


```python
from machine import MachineA, MachineB

class MachineType(Enum):
    MachineA = 0,
    MachineB = 1


class MachineFactory(AbstractFactory):

    def __init__(self):
        pass

    def create_object(self, type, level):
        if(type == MachineType.MachineA):
            return MachineA(level)
        elif(type == MachineType.MachineB):
            return MachineB(level)
        else:
            assert 0, "Invalid Machine Type"
```

Fig 6 Machine Factory

After considering this machine product, abstract factory, machine, factory and product factory, we can switch to the main program. We create the machine factory and the product factory. The two factories are initialized here using these two factories which are created here, we can create the objects.

In this list, first of all, we create an object of type machine A using the machine factory with a level zero. In the second statement, we create a machine B type object with a level one using the machine factory.



```python
from machineFactory import MachineType, MachineFactory
from productFactory import ProductType, ProductFactory


if __name__ == "__main__":
    mFactory = MachineFactory()
    pFactory = ProductFactory()

    resources = [
        mFactory.create_object(MachineType.MachineA, 0),
        mFactory.create_object(MachineType.MachineB, 1),
        pFactory.create_object(ProductType.ProductA, 0),
        pFactory.create_object(ProductType.ProductB, 1)
    ]
    for res in resources:
        res.work()
```

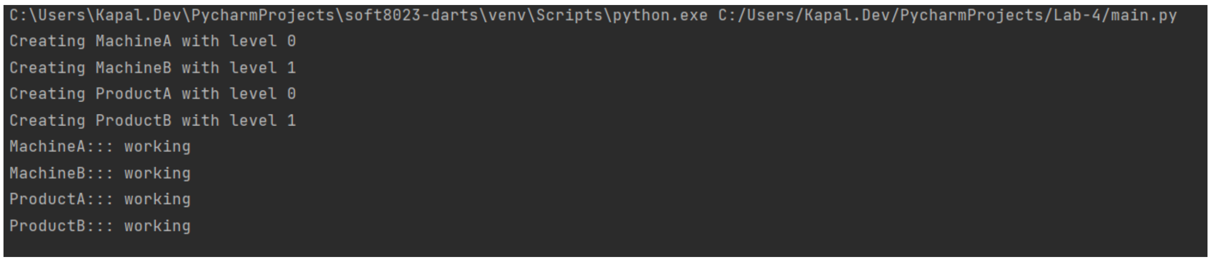
Fig 7 Main Program

Similar to that, we create a product with the level zero using the product factory. And again, we create product to be with the level one using the product factory. Once you have created all these objects using the factories, you can ask all the resources to work because if you remember, this work method was there in all the machines and product.

So whatever method has been created by the factory, they need to provide this method of work. So irrespective of the machine or product we can call the method of work on this particular resource right now, let's try to run main.py program.

Here we see that first, when we created a machine with a level zero, this particular statement is printed, that machine is created with the level zero similar to that machine being created with level one product is created with level zero product being created with level one.

And using this loop, we can ask all the resources to work. So that is why we are getting these statements. Machine A, working machine B, working product A, working product B.



```
C:\Users\Kapal.Dev\PycharmProjects\soft8023-darts\venv\Scripts\python.exe C:/Users/Kapal.Dev/PycharmProjects/Lab-4/main.py
Creating MachineA with level 0
Creating MachineB with level 1
Creating ProductA with level 0
Creating ProductB with level 1
MachineA::: working
MachineB::: working
ProductA::: working
ProductB::: working
```

Fig 8 Output

Exercise

Discuss any five examples of abstract factory design pattern and implement any two of them.