

**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 | **Mirza Sohaib Baig** |
| Student ID | **20B-041-SE** |
| Date | **18-11-2022** |

**Assessment**

|  |  |
| --- | --- |
| Marks Obtained |  |
| Remarks |  |
| Signature |  |

**Usman Institute of Technology**

**Department of Computer Science**

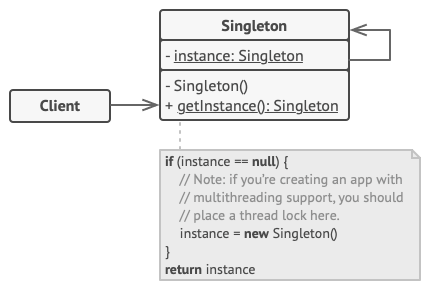
**SE308 - Software Design and Architecture**

**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



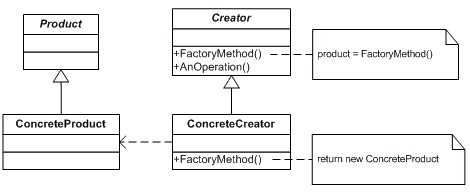
class Singleton: **Output**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | \_\_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 | |  |

## 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** o defines the interface of objects the factory method creates
* **ConcreteProduct** o implements the Product interface
* **Creator** o 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.

o may call the factory method to create a Product object.

* **ConcreteCreator** o overrides the factory method to return an instance of a ConcreteProduct.

### Example in 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'**

):

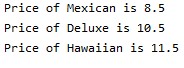
print

(

**'Price of {0} is {1}'**

.format(pizza\_type,

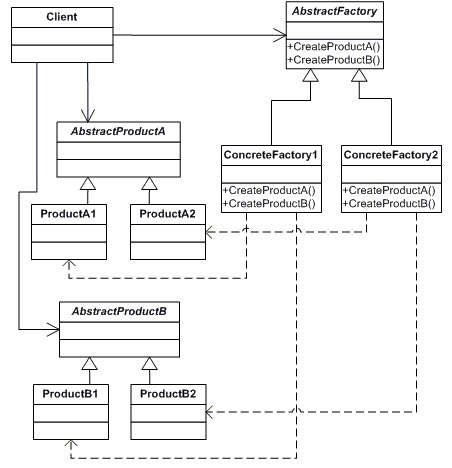
PizzaFactory.create\_pizza(pizza\_type).get\_price()))



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

### Example in 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()     |  | | --- | | I am a wooden door  I can only fit wooden doors  I am an iron door  I can only fit iron doors |   door = woodenFactory.makeDoor()  expert = woodenFactory.makeFittingExpert()  door.getDescription() expert.getDescription()  ironFactory = IronDoorFactory()    door = ironFactory.makeDoor()  expert = ironFactory.makeFittingExpert()  door.getDescription() expert.getDescription() |

**Student** **Tasks**:

## Class Task

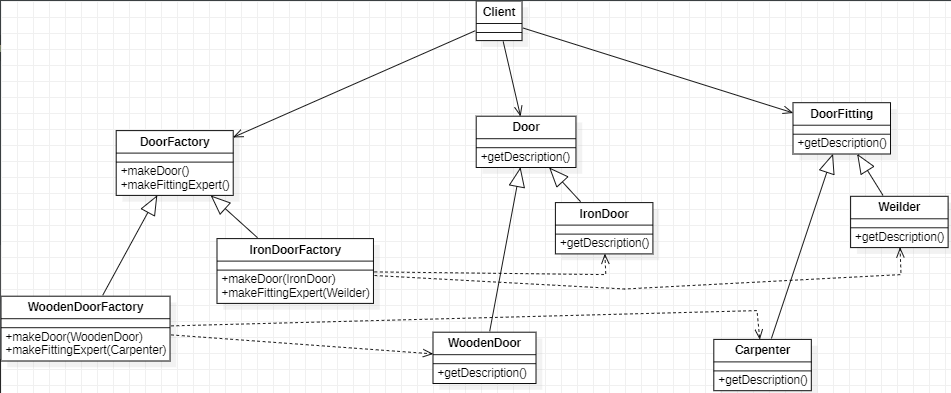
1. For Factory Pattern, Abstract Factory Pattern

a. Generate (from StarUML) UML diagram of the above patterns

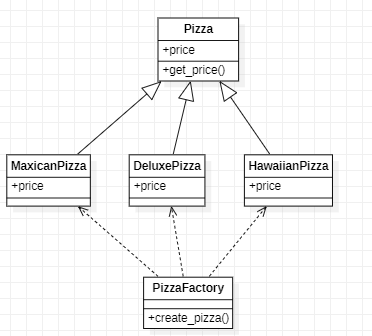
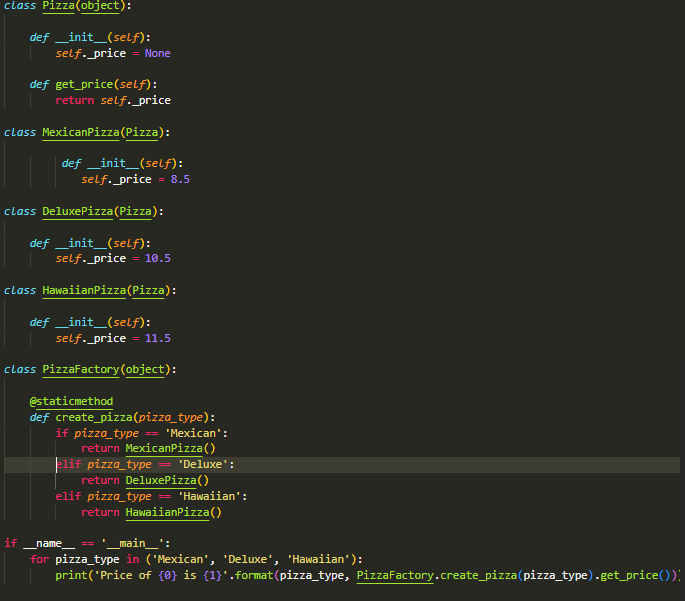
Tools - > Apply Pattern - > Pattern Repository -> GoF

1. Compare your generated UML diagram with given code (example in python)
2. Convert your generated UML diagram according to the given code
3. Run the code and analyze the output

**Abstract Factory Method**



**Factory Method**



## Home Task

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

