**ABSTRACT FACTORY**

from \_\_future\_\_ import annotations

from abc import ABC, abstractmethod

class AbstractFactory(ABC):

@abstractmethod

def createChair(self)->Chair:

pass

@abstractmethod

def createTable(self)->Table:

pass

class ModernFactory(AbstractFactory):

def createChair(self)->Chair:

return ModernChair()

def createTable(self)->Table:

return ModernTable()

class VictorianFactory(AbstractFactory):

def createChair(self)->Chair:

return VictorianChair()

def createTable(selg)->Table:

return VictorianTable()

class Chair(ABC):

@abstractmethod

def function\_chair(self)->str:

pass

class ModernChair(Chair):

def function\_chair(self)->str:

return "I am MODERN CHAIR"

class VictorianChair(Chair):

def function\_chair(self)->str:

return "I am VICTORIAN CHAIR"

class Table(ABC):

@abstractmethod

def function\_table(self)->str:

pass

class ModernTable(Table):

def function\_table(self)->str:

return "I AM MODERN TABLE"

class VictorianTable(Table):

def function\_table(self)->str:

return "I AM VICTORIAN TABLE"

def client\_code(factory: AbstractFactory)->None:

product\_chair = factory.createChair()

product\_table = factory.createTable()

print(f"{product\_chair.function\_chair()}")

print(f"{product\_table.function\_table()}")

if \_\_name\_\_ == "\_\_main\_\_":

print("Test Tip 1")

client\_code(ModernFactory())

print("\n")

print("Test Tip 2")

client\_code(VictorianFactory())

**BUILDER**

from \_\_future\_\_ import annotations

from abc import ABC, abstractmethod

from typing import Any

class Builder(ABC):

@property

@abstractmethod

def product(self) -> None:

pass

@abstractmethod

def setSeats(self) -> None:

pass

@abstractmethod

def setEngine(self) -> None:

pass

@abstractmethod

def setGPS(self) -> None:

pass

class CarBuilder(Builder):

def \_\_init\_\_(self)->None:

self.reset()

#o instanta noua de builder trebuie sa contina un obj blank, care va fi folosit ulterior in asamblare

def reset(self)->None:

self.\_product=Car()

def setSeats(self)->None:

self.\_product.add("Am adaugat rotile.... CarBuilder\n")

def setEngine(self)->None:

self.\_product.add("Am Adaugat Motorul .... CarBuilder\n")

def setGPS(self)->None:

self.\_product.add("Am Adaugat GPS .... CarBuilder\n")

@property

def product(self)->Car: #un fel de getter

product=self.\_product

self.reset()

return product

class CarManualBuilder(Builder):

def \_\_init\_\_(self)->None:

self.reset()

def reset(self)->None:

self.\_product=ManualCar()

def setSeats(self) -> None:

self.\_product.add("Am adaugat Rotile ... Manual Car\n")

def setEngine(self)->None:

self.\_product.add("Am Adaugat Motorul .... Manual car\n")

def setGPS(self)->None:

self.\_product.add("Am Adaugat GPS .... Manual car\n")

@property

def product(self)->ManualCar:

product=self.\_product

self.reset()

return product

class Car():

def \_\_init\_\_(self)->None:

self.parts=[]

def add (self,part:Any)->None:

self.parts.append(part)

def list\_parts(self)->None:

print(f"Masina (Car) este facuta astfel: {', '.join(self.parts)}", end="")

class ManualCar():

def \_\_init\_\_(self)->None:

self.parts=[]

def add (self,part:Any)->None:

self.parts.append(part)

def list\_parts(self)->None:

print(f"Masina (ManualCar) este facuta astfel: {', '.join(self.parts)}", end="")

class Director:

def \_\_init\_\_(self)->None:

self.\_builder=None

@property

def builder(self)->Builder: # private filedul ????

return self.\_builder

@builder.setter

def builder(self,builder:Builder)->None: #aici dam tipul de builder car sau manual car la apel

self.\_builder=builder

def build\_minimal\_car(self) -> None:

self.builder.setSeats()

self.builder.setEngine()

def build\_cool\_car(self) -> None:

self.builder.setSeats()

self.builder.setEngine()

self.builder.setGPS()

if \_\_name\_\_ == "\_\_main\_\_":

"""

The client code creates a builder object, passes it to the director and then

initiates the construction process. The end result is retrieved from the

builder object.

"""

director = Director()

builder1 = CarBuilder()

director.builder = builder1

print("Standard basic product: ")

director.build\_minimal\_car()

builder1.product.list\_parts()

print("\n")

print("Standard full featured product: ")

director.build\_cool\_car()

builder1.product.list\_parts()

print("\n")

print("\*\*\*\*\*MANUAL\*\*\*\*\*\*")

builder2=CarManualBuilder()

director.builder=builder2

director.build\_cool\_car()

builder1.product.list\_parts()

print("\n")

# Remember, the Builder pattern can be used without a Director class.

print("Custom product: ")

builder1.setSeats()

builder1.setEngine()

builder1.product.list\_parts()

**FACTORY METHOD**

from \_\_future\_\_ import annotations

from abc import ABC, abstractmethod

class Dialog(ABC):

@abstractmethod

def createButton(self):

pass

def render(self)->str:

product = self.createButton()

rez= f"Am apelat metoda render cu {product.operation()}"

return reZ

class WindowsDialog(Dialog):

"""

Note that the signature of the method still uses the abstract product type,

even though the concrete product is actually returned from the method. This

way the Dialog can stay independent of concrete product classes.

"""

def createButton(self) -> Button:

return WindowsButton()

class WebDialog(Dialog):

def createButton(self) -> Button:

return HTMLButton()

class Button(ABC):

"""

The Button interface declares the operations that all concrete Buttons

must implement.

"""

@abstractmethod

def operation(self) -> str:

pass

"""

Concrete Buttons provide various implementations of the Button interface.

"""

class WindowsButton(Button):

def operation(self) -> str:

return "{Result of the WindowsButton}"

class HTMLButton(Button):

def operation(self) -> str:

return "{Result of the HTMLButton}"

def client\_code(Dialog: Dialog) -> None:

"""

The client code works with an instance of a concrete Dialog, albeit through

its base interface. As long as the client keeps working with the Dialog via

the base interface, you can pass it any Dialog's subclass.

"""

print(f"Client: I'm not aware of the Dialog's class, but it still works.\n"

f"{Dialog.render()}", end="")

if \_\_name\_\_ == "\_\_main\_\_":

print("App: Launched with the WindowsDialog.")

client\_code(WindowsDialog())

print("\n")

print("App: Launched with the ConcreteDialog2.")

client\_code(WebDialog())

**#NAIVE Singleton**

class SingletonMeta(type):

\_instances={}

def \_\_call\_\_(cls,\*args,\*\*kwargs):

if cls not in cls.\_instances:

instance=super().\_\_call\_\_(\*args,\*\*kwargs)

cls.\_instances[cls]=instance

return cls.\_instances[cls]

class Singleton(metaclass=SingletonMeta):

def function(self):

print ("Some Business Logic")

if \_\_name\_\_=="\_\_main\_\_":

s1=Singleton()

s2=Singleton()

if id(s1)==id(s2):

print("ok")

else:

print("N o mers")

**#thread safe**

from threading import Lock, Thread

class SingletonMeta(type):

"""

This is a thread-safe implementation of Singleton.

"""

\_instances = {}

\_lock: Lock = Lock()

"""

We now have a lock object that will be used to synchronize threads during

first access to the Singleton.

"""

def \_\_call\_\_(cls, \*args, \*\*kwargs):

"""

Possible changes to the value of the `\_\_init\_\_` argument do not affect

the returned instance.

"""

# Now, imagine that the program has just been launched. Since there's no

# Singleton instance yet, multiple threads can simultaneously pass the

# previous conditional and reach this point almost at the same time. The

# first of them will acquire lock and will proceed further, while the

# rest will wait here.

with cls.\_lock:

# The first thread to acquire the lock, reaches this conditional,

# goes inside and creates the Singleton instance. Once it leaves the

# lock block, a thread that might have been waiting for the lock

# release may then enter this section. But since the Singleton field

# is already initialized, the thread won't create a new object.

if cls not in cls.\_instances:

instance = super().\_\_call\_\_(\*args, \*\*kwargs)

cls.\_instances[cls] = instance

return cls.\_instances[cls]

class Singleton(metaclass=SingletonMeta):

value: str = None

"""

We'll use this property to prove that our Singleton really works.

"""

def \_\_init\_\_(self, value: str) -> None:

self.value = value

def some\_business\_logic(self):

"""

Finally, any singleton should define some business logic, which can be

executed on its instance.

"""

def test\_singleton(value: str) -> None:

singleton = Singleton(value)

print(singleton.value)

if \_\_name\_\_ == "\_\_main\_\_":

# The client code.

print("If you see the same value, then singleton was reused (yay!)\n"

"If you see different values, "

"then 2 singletons were created (booo!!)\n\n"

"RESULT:\n")

process1 = Thread(target=test\_singleton, args=("FOO",))

process2 = Thread(target=test\_singleton, args=("BAR",))

process1.start()

process2.start()

**PROTOTYPE**

from abc import ABC, abstractmethod

import time

import copy

import datetime

# Class Creation

class Prototype(ABC):

# Constructor:

def \_\_init\_\_(self):

# Mocking an expensive call

time.sleep(3)

# Base attributes

self.height = None

self.age = None

self.defense = None

self.attack = None

# Clone Method:

@abstractmethod

def clone(self):

pass

class Shopkeeper(Prototype):

def \_\_init\_\_(self,height,age,defense,attack):

super().\_\_init\_\_()

self.height = height

self.age = age

self.defense = defense

self.attack = attack

#specific

self.charisma=30

def clone(self):

return copy.deepcopy(self)

class Warrior(Prototype):

def \_\_init\_\_(self, height, age, defense, attack):

super().\_\_init\_\_()

self.height = height

self.age = age

self.defense = defense

self.attack = attack

self.stamina = 60

# Overwritting Cloning Method

def clone(self):

return copy.deepcopy(self)

class Mage(Prototype):

def \_\_init\_\_(self, height, age, defense, attack):

super().\_\_init\_\_()

self.height = height

self.age = age

self.defense = defense

self.attack = attack

self.mana = 100

# Overwritting Cloning Method

def clone(self):

return copy.deepcopy(self)

if \_\_name\_\_ == "\_\_main\_\_":

print('Starting to create a Shopkeeper NPC: ', datetime.datetime.now().time())

shopkeeper = Shopkeeper(180, 22, 5, 8)

print('Finished creating a Shopkeeper NPC: ', datetime.datetime.now().time())

print('Attributes: ' + ', '.join("%s: %s" % item for item in vars(shopkeeper).items()))

print('Instantiating 1000 NPCs: ', datetime.datetime.now().time())

shopkeeper\_template = Shopkeeper(180, 22, 5, 8)

warrior\_template = Warrior(185, 22, 4, 21)

mage\_template = Mage(172, 65, 8, 15)

for i in range(333):

shopkeeper\_clone = shopkeeper\_template.clone()

warrior\_clone = warrior\_template.clone()

mage\_clone = mage\_template.clone()

print(f'Finished creating NPC trio clone {i} at: ', datetime.datetime.now().time())

print('Finished instantiating NPC population at: ', datetime.datetime.now().time())

**ADAPTER**

import math

from operator import truediv

from xmlrpc.client import Boolean

class RoundPeg:

def \_\_init\_\_(self,radius):

self.radius=radius

def getRadius(self):

return self.radius

class RoundHole:

def \_\_init\_\_(self,radius):

self.radius=radius

def getRadius(self)->float:

return self.radius

def fits(self, \_peg:RoundPeg):

if self.getRadius() >= \_peg.getRadius():

print("true\n")

else:

print("False\n")

class SquarePeg:

def \_\_init\_\_(self,width):

self.width=width

def getWidth(self)->float:

return self.width

class SquarePegAdapter(RoundPeg):

def \_\_init\_\_(self, \_peg:SquarePeg):

self.peg=\_peg

def getRadius(self)->float:

return self.peg.getWidth()\*math.sqrt(2) /2

def main():

hole = RoundHole(5)

rpeg= RoundPeg(5)

hole.fits(rpeg)

small\_sqpeg= SquarePeg(5)

large\_sqpeg= SquarePeg(10)

#hole.fits(small\_sqpeg) #incopatibil

small\_sqpeg\_adapter=SquarePegAdapter(small\_sqpeg)

large\_sqpeg\_adapter=SquarePegAdapter(large\_sqpeg)

hole.fits(small\_sqpeg\_adapter)

hole.fits(large\_sqpeg\_adapter)

if \_\_name\_\_=="\_\_main\_\_":

main()