

**Usman Institute of Technology**

**Department of Computer Science Fall 2022**

Name: Muhammad Waleed

Roll no: 20B-115-SE

Course: Software Design and Architecture (SE-308)

Course Instructor: Misbah ud Din

Date: 1-Dec-2022

# Lab Tasks:

## ObserverDesignPattern:

Diagram

Description automatically generated

Code:

from abc import ABC, abstractmethod

class Publisher:

    def \_\_init\_\_(self):

        self.\_\_subscribers = []

        self.\_\_content = None

    def attach(self, subscriber):

        self.\_\_subscribers.append(subscriber)

    def detach(self):

        self.\_\_subscribers.pop()

    def get\_subscribers(self):

        return [type(x).\_\_name\_\_

        for x in self.\_\_subscribers]

    def updateSubscribers(self):

        for sub in self.\_\_subscribers:

            sub.update()

    def add\_content(self, content):

        self.\_\_content = content

    def get\_content(self):

        return ("Content:" + self.\_\_content)

class Subscriber(ABC):

    @abstractmethod

    def update(self):

        pass

class SiteSubscriber(Subscriber):

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

        self.publisher = publisher

        self.publisher.attach(self)

    def update(self):

        print(type(self).\_\_name\_\_,

        self.publisher.get\_content())

# -------------------- # Subscriber 2

class IntranetSubscriber(Subscriber):

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

        self.publisher = publisher

        self.publisher.attach(self)

    def update(self):

        print(type(self).\_\_name\_\_,

        self.publisher.get\_content())

# -------------------- # Subscriber 3

class ApiSubscriber(Subscriber):

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

        self.publisher = publisher

        self.publisher.attach(self)

    def update(self):

        print(type(self).\_\_name\_\_,

        self.publisher.get\_content())

publisher = Publisher()

for subs in [SiteSubscriber, IntranetSubscriber, ApiSubscriber]:

    subs(publisher)

print("All Subscriber: ", publisher.get\_subscribers())

print("------------------------------------------------")

publisher.add\_content('Update content on all subscribers.')

publisher.updateSubscribers()

print("------------------------------------------------")

publisher.detach()

print("Remaining Subscriber: ", publisher.get\_subscribers())

print("------------------------------------------------")

publisher.add\_content('Updated content for remaining subscriber.')

publisher.updateSubscribers()

Ouput:

Text

Description automatically generated

## CommandDesignPattern:

Diagram

Description automatically generated

Code:

from abc import ABC, abstractmethod

class Order(ABC):

    @abstractmethod

    def process(self):

        pass

class BuyStock(Order):

    def \_\_init\_\_(self, stock):

        self.stock = stock

    def process(self):

        self.stock.buy()

class SellStock(Order):

    def \_\_init\_\_(self, stock):

        self.stock = stock

    def process(self):

        self.stock.sell()

class Trade:

    def buy(self):

        print("Stock buy request placed.")

    def sell(self):

        print("Stock sell request placed.")

class Invoker:

    def \_\_init\_\_(self):

        self.\_queue = []

    def process\_order(self, order):

        self.\_queue.append(order)

        order.process()

trade = Trade()

buy\_stock = BuyStock(trade)

sell\_stock = SellStock(trade)

invoker = Invoker()

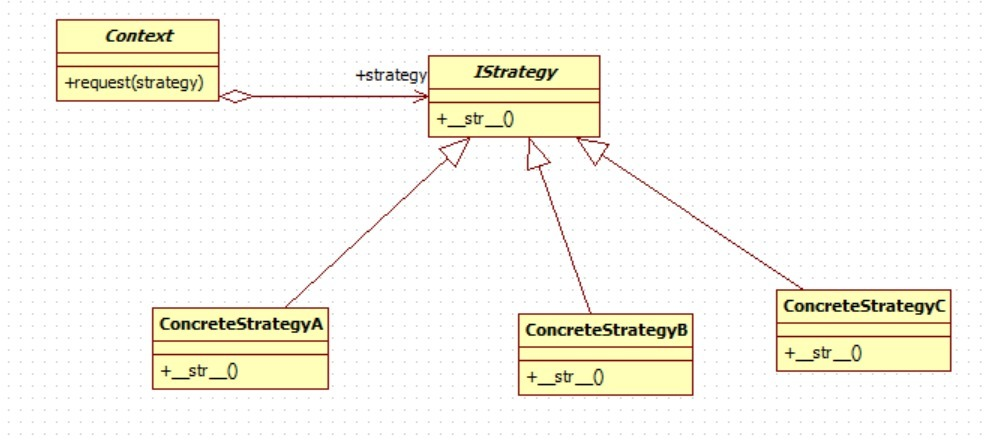
invoker.process\_order(buy\_stock)

invoker.process\_order(sell\_stock)

Output:



## StrategyDesignPattern:



Code:

from abc import ABCMeta, abstractmethod

class Context():

    # "This is the object whose behavior will change"

    @staticmethod

    def request(strategy):

    # "The request is handled by the class passed in"

        return strategy()

class IStrategy(metaclass=ABCMeta):

    "A strategy Interface"

    @staticmethod

    @abstractmethod

    def \_\_str\_\_():

        return "I am a Strategy"

class ConcreteStrategyA(IStrategy):

    "A Concrete Strategy Subclass"

    def \_\_str\_\_(self):

        return "I am ConcreteStrategyA"

class ConcreteStrategyB(IStrategy):

    "A Concrete Strategy Subclass"

    def \_\_str\_\_(self):

        return "I am ConcreteStrategyB"

class ConcreteStrategyC(IStrategy):

    "A Concrete Strategy Subclass"

    def \_\_str\_\_(self):

        return "I am ConcreteStrategyC"

# The Client

CONTEXT = Context()

print(CONTEXT.request(ConcreteStrategyA))

print(CONTEXT.request(ConcreteStrategyB))

print(CONTEXT.request(ConcreteStrategyC))

Output:

Text

Description automatically generated

# Home Tasks:

## Sorting Strategy:

#strategy design pattern example

class SortStrategy:

    def sort(self, data):

        raise NotImplementedError

class QuickSort(SortStrategy):

    def sort(self, data):

        print('Quick sort')

class MergeSort(SortStrategy):

    def sort(self, data):

        print('Merge sort')

class Sorter:

    def \_\_init\_\_(self, sort\_strategy):

        self.sort\_strategy = sort\_strategy

    def sort(self, data):

        self.sort\_strategy.sort(data)

if \_\_name\_\_ == '\_\_main\_\_':

    quick = QuickSort()

    merge = MergeSort()

    sorter = Sorter(quick)

    sorter.sort([1, 2, 3, 4, 5])

    sorter.sort\_strategy = merge

    sorter.sort([1, 2, 3, 4, 5])

Output:

Graphical user interface, text, application

Description automatically generated

Youtube Observer:

class YouTubeChannel:

    def \_\_init\_\_(self):

        self.subscribers = []

        self.video = None

    def attach(self, subscriber):

        self.subscribers.append(subscriber)

    def detach(self, subscriber):

        self.subscribers.remove(subscriber)

    def notify(self):

        for subscriber in self.subscribers:

            subscriber.update()

    def upload(self, video):

        self.video = video

        self.notify()

class Subscriber:

    def \_\_init\_\_(self, name):

        self.name = name

    def update(self):

        print(f'{self.name} has been notified of a new video')

yt = YouTubeChannel()

s1 = Subscriber('Waleed')

s2 = Subscriber('Bajwa')

s3 = Subscriber('Farhan')

s4 = Subscriber('Huzzu')

yt.attach(s1)

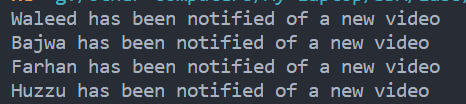
yt.attach(s2)

yt.attach(s3)

yt.attach(s4)

yt.upload('Python Tutorial')

Output:



## FoodCommand:

from abc import ABC, abstractmethod

class Food:

    @abstractmethod

    def cook(self):

        pass

class Pizza(Food):

    def cook(self):

        print('Pizza is cooking')

class Burger(Food):

    def cook(self):

        print('Burger is cooking')

class FoodCommand:

    def \_\_init\_\_(self, food):

        self.food = food

    def execute(self):

        self.food.cook()

class Waiter:

    def \_\_init\_\_(self):

        self.\_\_commands = []

    def add\_command(self, command):

        self.\_\_commands.append(command)

    def remove\_command(self, command):

        self.\_\_commands.remove(command)

    def execute\_commands(self):

        for command in self.\_\_commands:

            command.execute()

if \_\_name\_\_ == '\_\_main\_\_':

    pizza = Pizza()

    burger = Burger()

    pizza\_command = FoodCommand(pizza)

    burger\_command = FoodCommand(burger)

    waiter = Waiter()

    waiter.add\_command(pizza\_command)

    waiter.add\_command(burger\_command)

    waiter.execute\_commands()

Output:

Graphical user interface

Description automatically generated