Usman Institute of Technology

Department of Computer Science

Course Code: SE308

Course Title: Software Design and Architecture

Fall 2022

Lab 08

OBJECTIVE: Working on Design Patterns Contd.

- To Understand Behavioral Design Patterns.
- To implement Command, Observer & Strategy Design Patterns

Student Information

Student Name	
Student ID	
Date	

Assessment

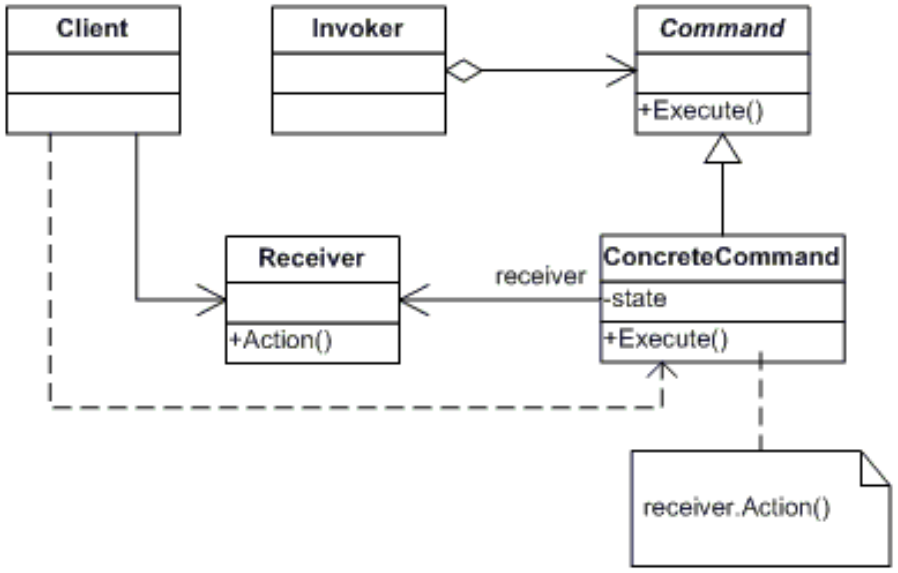
Marks Obtained	
Remarks	
Signature	

Usman Institute of Technology
Department of Computer Science
SE308 - Software Design and Architecture
Lab 07

Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

UML class diagram



Class Diagram of Command Method

Participants

The classes and objects participating in this pattern are:

- **Command**
  - declares an interface for executing an operation.
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receive.
- **Client** (Application)
  - creates a ConcreteCommand object and sets its receiver.
- **Invoker** (MenuItem)
  - asks the command to carry out the request.

Example in Python

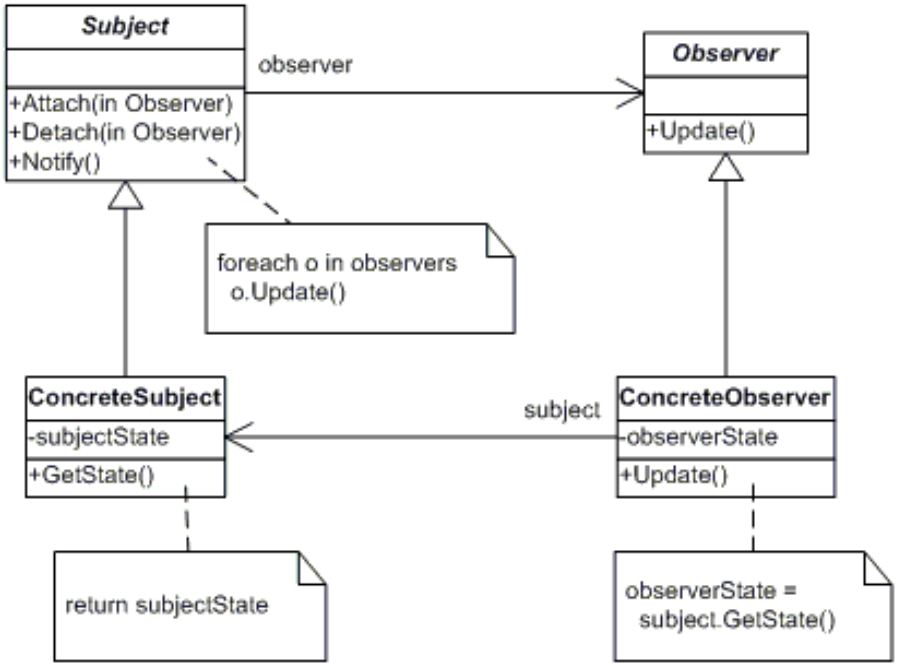
```
from abc import ABC, abstractmethod

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)
```

```
Stock buy request placed.
Stock sell request placed.
```

Observer Method

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

UML class diagram



Class Diagram of Observer Method

Participants

- **Subject**
  - knows its observers. Any number of Obsever objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **Concrete Subject**
  - stores state of interest to Concrete Observer objects.
  - sends a notification to its observes when its state changes.
- **Concrete Observer**
  - maintains a reference to a Concrete Subject object.
  - stores state that should stay consistent with the subjects.
  - implements the Observer updating interface to keep its state consistent with the subject's.

Example in Python

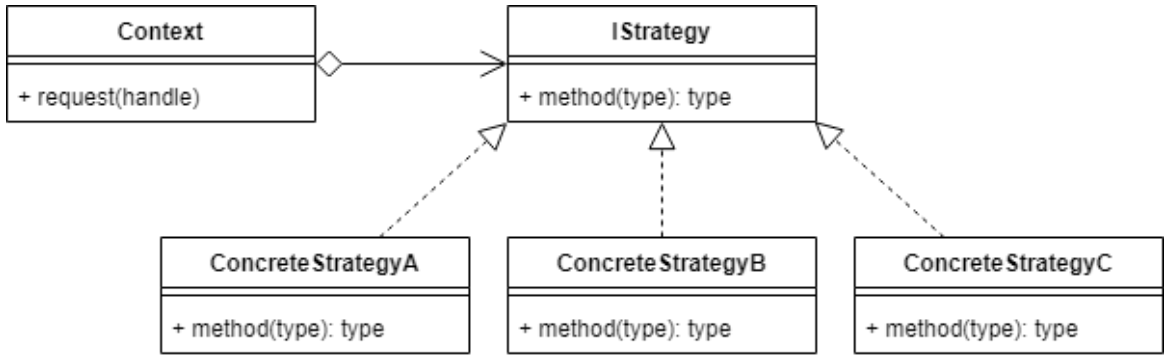
```
from abc import ABC, abstractmethod

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()
```

Strategy Method

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

UML class diagram



Class Diagram of Strategy Method

Participant

The classes and objects participating in this pattern are:

- **Strategy Interface**: An interface that all Strategy subclasses/algorithms must implement.
- **Concrete Strategy**: The subclass that implements an alternative algorithm.
- **Context**: This is the object that receives the concrete strategy in order to execute it.

Example in Python

```
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__():
        "Implement the __str__ dunder"

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))
```

```
python ./strategy/strategy_concept.py
I am ConcreteStrategyA
I am ConcreteStrategyB
I am ConcreteStrategyC
```

Student Tasks:

Class Task

For Adaptor, Bridge & Composite Pattern

a) Generate UML diagram (from StarUML) of the above patterns
b) Compare your generated UML diagram with given code (example in python)
c) Convert your generated UML diagram according to the given code
d) Run the code and analyze the output

Home Task

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