

**Usman Institute of Technology**

**Department of Computer Science**

**Course Code: SE308**

**Course Title: Software Design and Architecture**

# Summer 2024

**Lab 08**

**OBJECTIVE: Working on Design Patterns Contd.**

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

## Student Information

|  |  |
| --- | --- |
| Student Name | **Jawwad Bhatti** |
| Student ID | **20B-011-SE** |
| Date | **23-07-2024** |

## Assessment

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

**Usman Institute of Technology**

**Department of Computer Science**

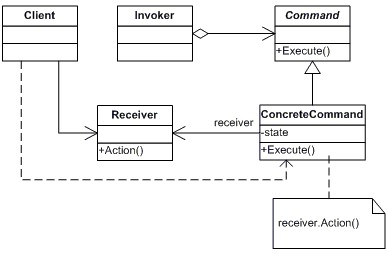
**SE308 - Software Design and Architecture**

**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** (Menultem)

- asks the command to carry out the request.

## Example in Python

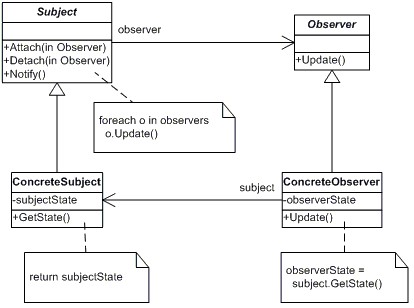
|  |
| --- |
| from abc import ABC, abstractmethod  class Order(ABC): @abstractmethod def process(self):  pass  class BuyStock(Order): def \_\_init\_\_(self, stock):  self.stock = stock def process(self): self.stock.buy()  class SellStock(Order): def \_\_init\_\_(self, stock):  self.stock = stock def 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) |



### 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.

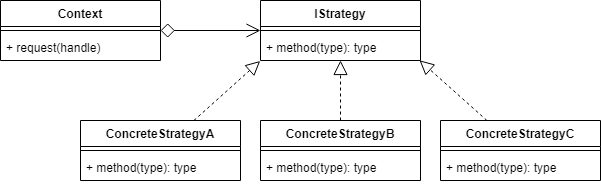
## Example in Python

|  |
| --- |
| 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

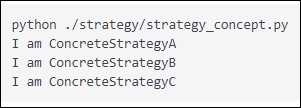
.

request

(

ConcreteStrategyC

))



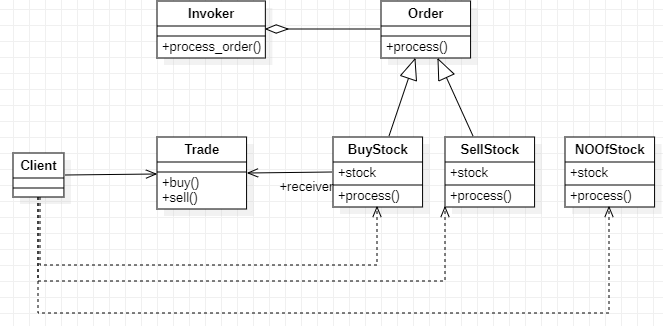
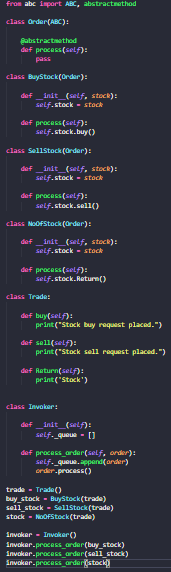
**Student** **Tasks**:

## Class Task

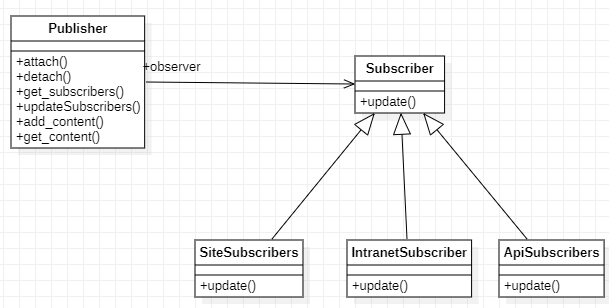
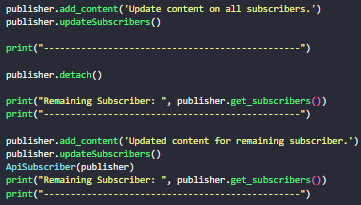
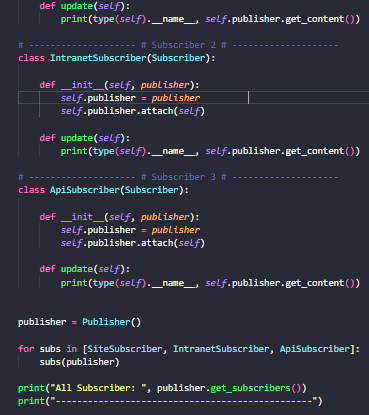
For Command, Observer & Strategy Pattern

1. Generate UML diagram (from StarUML) of the above patterns
2. Compare your generated UML diagram with given code (example in python)
3. Convert your generated UML diagram according to the given code
4. Run the code and analyze the output

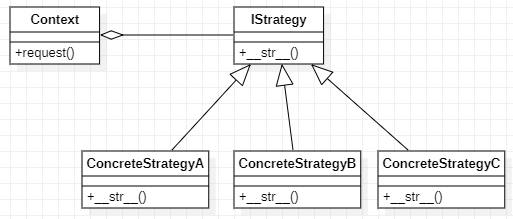
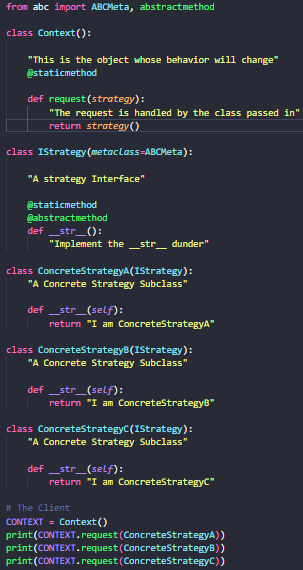
***Command***



***Observer***



***Strategy***



## Home Task

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

