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 Structural Design Patterns.
- To implement Adaptor, Bridge & Composite Design Patterns

Student Information

Student Name	
Student ID	
Date	

Assessment

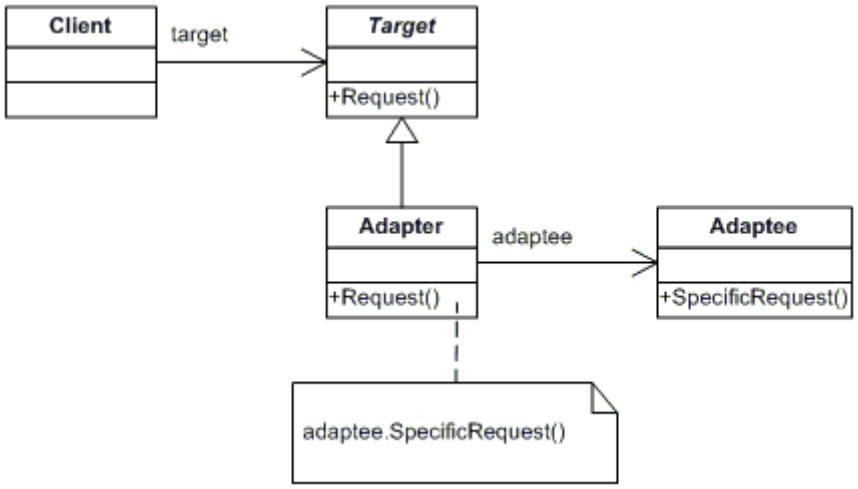
Marks Obtained	
Remarks	
Signature	

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

Adapter

"Convert the interface of a class into another interface clients expect."

UML class diagram



Class Diagram of Adapter Method

Participants

The classes and objects participating in this pattern are:

Target

- defines the domain - specific interface that Client uses.

Client

- collaborates with objects conforming to the Target interface

Adaptee

- defines an existing interface that need adapting.

Adapter

- adapts the interface of Adaptee to the Target interface.

Example in Python

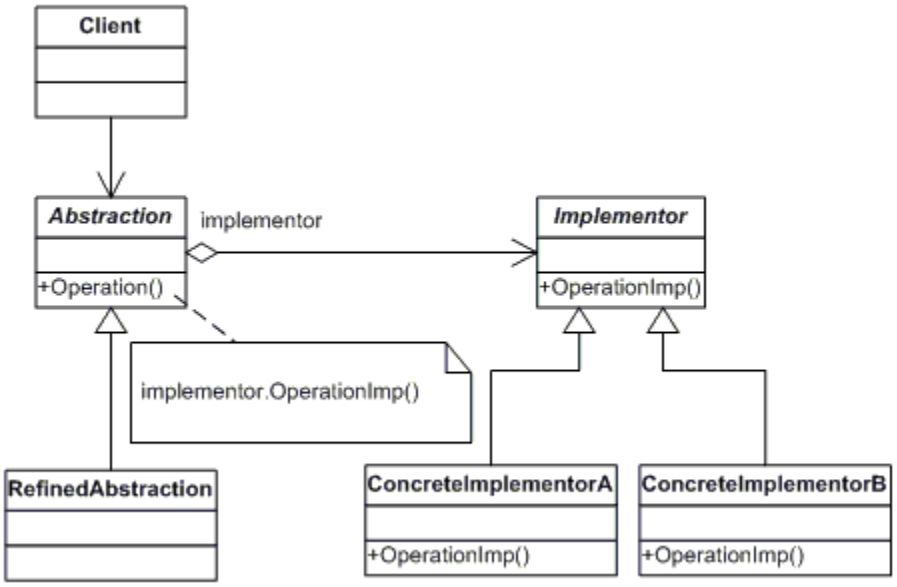
```
 from abc import ABC

class AbsAddress(ABC):
    line: str
    city: str
    country: str
    pin: str

class VendorAddress:
    def __init__(self, line1, line2, line3, city, country, pin):
        self.line1 = line1
        self.line2 = line2
        self.line3 = line3
        self.city = city
        self.country = country
        self.pin = pin

class CustomerAddress(AbsAddress):
    def __init__(self, line, city, country, pin):
        self.line = line
        self.city = city
        self.country = country
        self.pin = pin

class VendorAddressAdapter:
    def __init__(self, vendor_address):
        self.line = f'{vendor_address.line1}, {vendor_address.line2},
{vendor_address.line3}'
        self.city = vendor_address.city
        self.country = vendor_address.country
        self.pin = vendor_address.pin


# client
def print_address(address):
    print(f'{address.line}, {address.city}, {address.country}, {address.pin}')


if __name__ == '__main__':
   cust_address = CustomerAddress("Street 7", "A. B C Road", "Karachi", 74550)
   vend_address = VendorAddress("Home # 1", "Apartment 1", "Street 4", "A. B C Road",
"karachi", 45700)
    vend_address_adapt = VendorAddressAdapter(vend_address)

    for address in [address1, vend_address_adapt]:
        print_address(address)
```

```
Street 7, A. B C Road, Karachi, 74550
Home # 1, Apartment 1, Street 4, A. B C Road, karachi, 45700
```

Bridge Method

"Decouple an abstraction from its implementation so that the two can vary independently"

UML class diagram



Class Diagram of Bridge Method

Participants

The classes and objects participating in this pattern are:

Abstraction (BusinessObject)
- defines the abstraction's interface.
- maintains a reference to an object of type Implementor.

RefinedAbstraction (CustomersBusinessObject)
- extends the interface defined by Abstraction.

Implementor (DataObject)
- defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

ConcreteImplementor (CustomersDataObject)
- implements the Implementor interface and defines its concrete implementation.

Example in Python

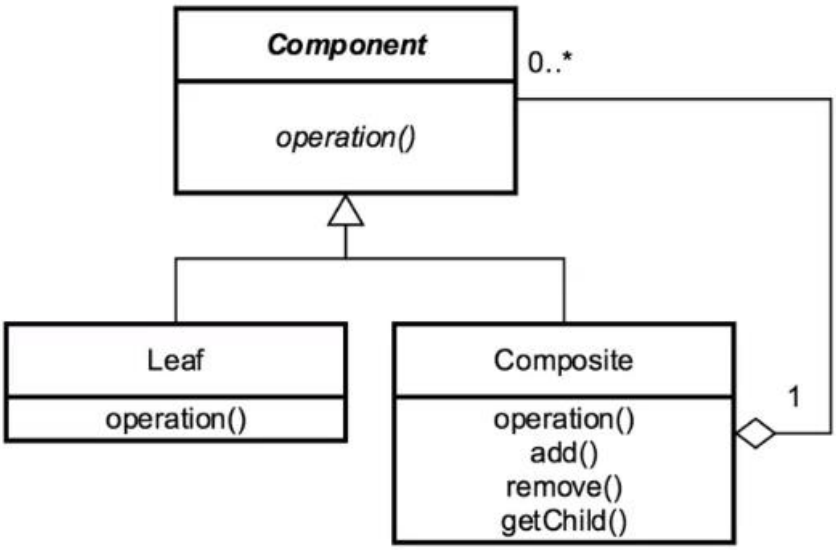
```
 class WebPage:
    def __init__(self, theme):
        self.theme = theme

    def getContent(self):
        pass

class About(WebPage):
    _theme = None

    def __init__(self, theme):
        self.theme = theme

    def getContent(self):
        return "About page in " + self.theme.getColor()

class Careers(WebPage):
    _theme = None

    def __init__(self, theme):
        self.theme = theme

    def getContent(self):
        return "Careers page in " + self.theme.getColor()

class Theme:
    def getColor(self):
        pass

class DarkTheme(Theme):
    def getColor(self):
        return 'Dark Black'

class LightTheme(Theme):
    def getColor(self):
        return 'Off White'

class AquaTheme(Theme):
    def getColor(self):
        return 'Light Blue'

if __name__ == '__main__':

    darkTheme = DarkTheme()
    lightTheme = LightTheme()

    about = About(darkTheme)
    careers = Careers(darkTheme)

    aboutLight = About(lightTheme)
    careersLight = Careers(lightTheme)

    print (about.getContent())
    print (careers.getContent())

    print(aboutLight.getContent())
    print(careersLight.getContent())
```

```
About page in Dark Black
Careers page in Dark Black
About page in Off White
Careers page in Off White
```

Composite Method

"Compose objects into tree structures to represent part-whole hierarchies"

UML class diagram



Class Diagram of Composite Method

Participant

The classes and objects participating in this pattern are:

Component Interface:

The interface that all leaves and composites should implement.

Leaf:

A single object that can exist inside or outside a composite.

Composite:

A collection of leaves and/or other composites.

Client

Manipulate objects in the composition through the component interface

Example in Python

```python
from abc import ABC, abstractmethod

class BaseDepartment(ABC):
    @abstractmethod
    def __init__(self, num_of_employees):
        pass

    @abstractmethod
    def print_department(self):
        pass

class Accounting(BaseDepartment):
    def __init__(self, num_of_employees):
        self.num_of_employees = num_of_employees

    def print_department(self):
        print(f"Accounting employees: {self.num_of_employees}")

class Development(BaseDepartment):
    def __init__(self, num_of_employees):
        self.num_of_employees = num_of_employees

    def print_department(self):
        print(f"Development employees: {self.num_of_employees}")

class Management(BaseDepartment):
    def __init__(self, num_of_employees):
        self.num_of_employees = num_of_employees
        self.childs = []

    def print_department(self):
        print(f"Management base employees: {self.num_of_employees}")
        total_emp_count = self.num_of_employees
        for child in self.childs:
            total_emp_count += child.num_of_employees
            child.print_department()
        print(f'Total employees: {total_emp_count}')

    def add_child_dept(self, dept):
        self.childs.append(dept)

#
acc_dept = Accounting(200)
dev_dept = Development(500)

management_dept = Management(50)
management_dept.add(acc_dept)
management_dept.add(dev_dept)

# print dept
management_dept.print_department()
```

```
Management base employees: 50
Accounting employees: 200
Development employees: 500
Total employees: 750
```

Student Tasks:

Class Task

For Adaptor, Bridge & Composite Pattern

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