

**Usman Institute of Technology**

**Department of Computer Science**

**Course Code: SE308**

**Course Title: Software Design and Architecture**

# Summer 2024

**Lab 07**

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

* To Understand Structural Design Patterns.
* To implement Adaptor, Bridge & Composite Design Patterns

**Student Information**

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

**Assessment**

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

**Usman Institute of Technology**

**Department of Computer Science**

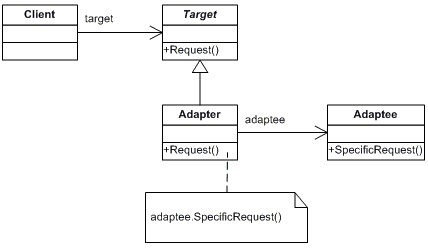
**SE308 - Software Design and Architecture**

**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.

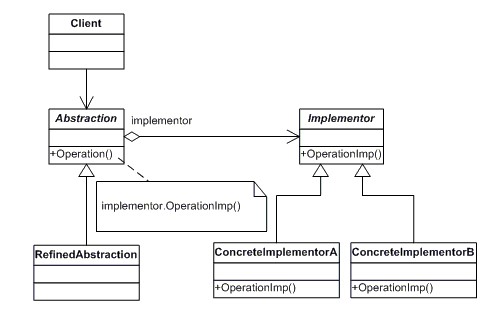
## Example in Python

|  |
| --- |
| 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) |



**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)** o defines the abstraction's interface.

o maintains a reference to an object of type Implementor.

**RefinedAbstraction**  **(CustomersBusinessObject)** o extends the interface defined by Abstraction.

**Implementor**  **(DataObject)** o 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)** o implements the Implementor interface and defines its concrete implementation.

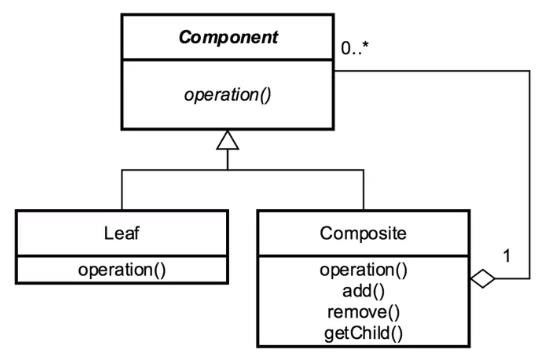
## Example in Python

|  |  |
| --- | --- |
| **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 page in Dark Black  Careers page in Dark Black  About page in Off White  Careers page in Off White |   about = About(darkTheme) careers = Careers(darkTheme)  aboutLight = About(lightTheme) careersLight = Careers(lightTheme)    print (about.getContent()) print (careers.getContent())  print(aboutLight.getContent()) print(careersLight.getContent()) |

## 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**:

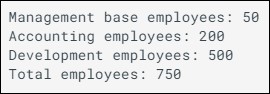
A collection of leaves and/or other composites.

**Client**

Manipulate objects in the composition through the component interface

**Example in Python**

|  |
| --- |
| 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() |



**Student** **Tasks**:

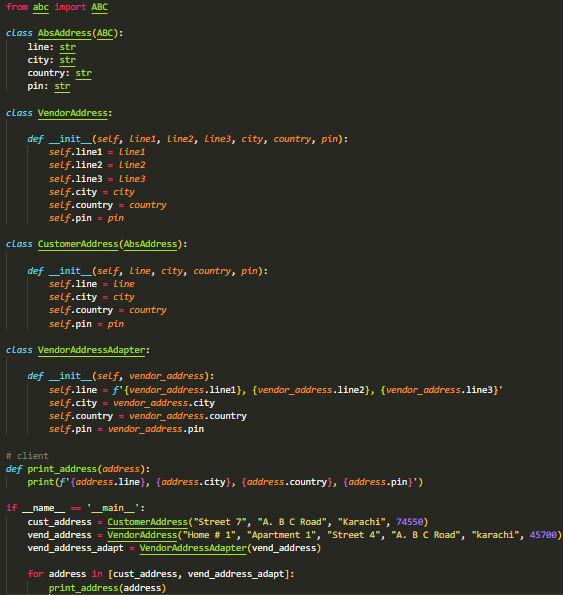
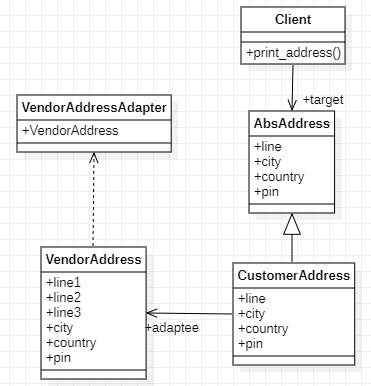
## Class Task

For Adaptor, Bridge & Composite Pattern

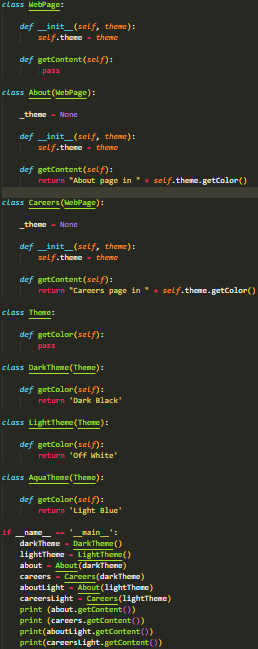
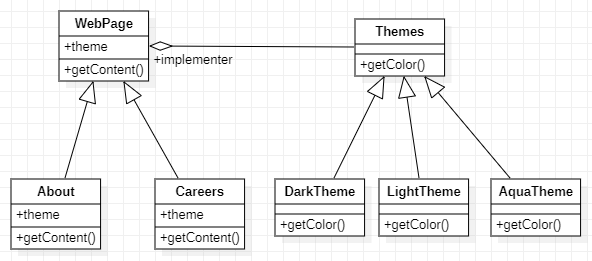
a. Generate (from StarUML) UML diagram of the above patterns

1. Compare your generated UML diagram with given code (example in python)
2. Convert your generated UML diagram according to the given code
3. Run the code and analyze the output

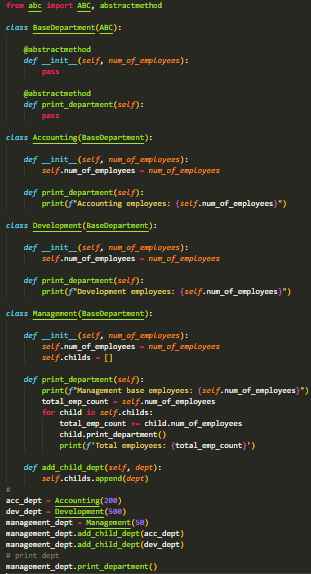
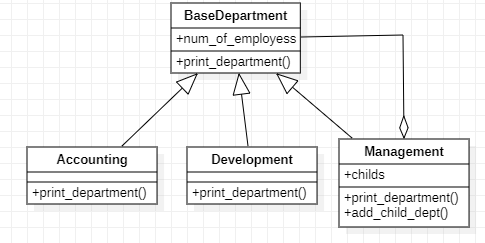
**Adapter**



**Bridge**



**Composite**



## Home Task

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

