**DESIGN PATTERNS**

Design patterns are used to represent the pattern used by developers to create software or web application. These patterns are selected based on the requirement analysis. The patterns describe the solution to the problem, when and where to apply the solution and the consequences of the implementation.

#### **Classification of Design Patterns**

1.Creational Design Patterns

2.Structural Design Patterns

3.Behavioral Design Patterns

1.Creational Design Patterns

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

1.Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

# Singleton Borg pattern

class Borg:

# state shared by each instance

\_\_shared\_state = dict()

# constructor method

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

self.\_\_dict\_\_ = self.\_\_shared\_state

self.state = 'GeeksforGeeks'

def \_\_str\_\_(self):

return self.state

# main method

if \_\_name\_\_ == "\_\_main\_\_":

person1 = Borg() # object of class Borg

person2 = Borg() # object of class Borg

person3 = Borg() # object of class Borg

person1.state = 'DataStructures' # person1 changed the state

person2.state = 'Algorithms' # person2 changed the state

print(person1) # output --> Algorithms

print(person2) # output --> Algorithms

person3.state = 'Geeks' # person3 changed the

# the shared state

print(person1) # output --> Geeks

print(person2) # output --> Geeks

print(person3) # output --> Geeks

2.Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

It provides one of the best ways to create an object.Objects are created without exposing the logic to client  
 Factory methods are very useful in adding new types of product without distributing the existing client code.

it avoids the tight coupling between the products and the creator classes and objects.

class Button(object):

html = ""

def get\_html(self):

return self.html

class Image(Button):

html = "<img></img>"

class Input(Button):

html = "<input></input>"

class Flash(Button):

html = "<obj></obj>"

class ButtonFactory():

def create\_button(self, typ):

targetclass = typ.capitalize()

return globals()[targetclass]()

button\_obj = ButtonFactory()

button = ['image', 'input', 'flash']

for b in button:

print button\_obj.create\_button(b).get\_html()

The button class helps to create the html tags and the associated html page. The client will not have access to the logic of code and the output represents the creation of html page.

The python code includes the logic of html tags, which specified value. The end user can have a look on the HTML file created by the Python code.

3.Builder Pattern

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

# concrete course

**class** Java():

# Class **for** Java Langauge

def fee\_structure(self):

self.fee\_structure = 10000

def batch\_availibility (self):

self.batches = 10

def \_\_str\_\_(self):

**return** "Java"

# concrete course

**class** Swift():

# **class** **for** Swift language"""

def fee\_structure(self):

self.fee\_structure = 5000

def batch\_availibility (self):

self.batches = 4

def \_\_str\_\_(self):

**return** "Swift"

# main method

**if** \_\_name\_\_ == "\_\_main\_\_":

j = Java() # object **for** Java

s = Swift() # object **for** Swift

print(f'Name of Course: {j} and its fee\_structure: {j.fee\_structure}')

print(f'Name of Course: {s} and its fee\_structure: {s.fee\_structure}')

2.Structural design pattern

Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

1.Decorator

Decorator Method is a [Structural Design Pattern](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/) which allows you to dynamically attach new behaviors to objects without changing their implementation by placing these objects inside the wrapper objects that contains the behaviors.

class WrittenText:

"""Represents a Written text """

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

self.\_text = text

def render(self):

return self.\_text

class UnderlineWrapper(WrittenText):

"""Wraps a tag in <u>"""

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

self.\_wrapped = wrapped

def render(self):

return "<u>{}</u>".format(self.\_wrapped.render())

class ItalicWrapper(WrittenText):

"""Wraps a tag in <i>"""

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

self.\_wrapped = wrapped

def render(self):

return "<i>{}</i>".format(self.\_wrapped.render())

class BoldWrapper(WrittenText):

"""Wraps a tag in <b>"""

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

self.\_wrapped = wrapped

def render(self):

return "<b>{}</b>".format(self.\_wrapped.render())

""" main method """

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

before\_gfg = WrittenText("Innovation Incubator")

after\_gfg = ItalicWrapper(UnderlineWrapper(BoldWrapper(before\_gfg)))

print("before :", before\_gfg.render())

print("after :", after\_gfg.render())

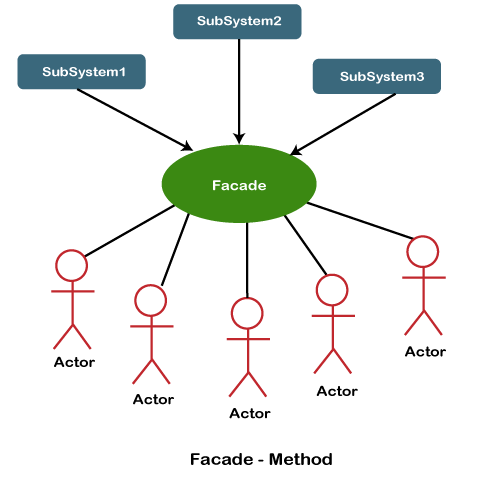
Output

before : Innovation Incubator

after : <i><u><b>Innovation Incubator</b></u></i>

2.Facade

Façade is a structural design pattern which provides the simpler interface to a library or complex set of classes. The word Façade is simply referred to an outer lying interface of complex patterns that contains several sub-systems.



"""Facade pattern with an example of Order pizza"""

**class** Ordering:

#Subsystem 1

def order(self):

print("Ordering")

**class** Preparing:

#Subsystem 2

def prepare(self):

print("Preparing...")

**class** Delivering:

#Subsystem 3

def deliver(self):

print("Delivering...")

**class** Operator:

'''Facade'''

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

self.ordering = Ordering()

self.preparing = Preparing()

self.delivering = Delivering()

def completeOrder(self):

self.ordering.order()

self.preparing.prepare()

self.delivering.deliver()

""" main method """

**if** \_\_name\_\_ == "\_\_main\_\_":

op = Operator()

op.completeOrder()

## **Behavioral Design Pattern:**

Behavioral patterns are all about identifying the common communication patterns between objects and realize these patterns

1.Chain of Responsibility

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.class ReportFormat(object):

PDF = 0

TEXT = 1

class Report(object):

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

self.title = 'Monthly report'

self.text = ['Things are going', 'really, really well.']

self.format\_ = format\_

class Handler(object):

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

self.nextHandler = None

def handle(self, request):

self.nextHandler.handle(request)

class PDFHandler(Handler):

def handle(self, request):

if request.format\_ == ReportFormat.PDF:

self.output\_report(request.title, request.text)

else:

super(PDFHandler, self).handle(request)

def output\_report(self, title, text):

print("<html>")

print("<head>")

print(" <title>%s</title>" % title)

print(" </head>'")

print("<body>")

for line in text:

print("<p>%s" % line)

print(" </body>'")

print("</html>")

class TextHandler(Handler):

def handle(self, request):

if request.format\_ == ReportFormat.TEXT:

self.output\_report(request.title, request.text)

else:

super(TextHandler, self).handle(request)

def output\_report(self, title, text):

print(5\*"\*" + title + 5\*"\*")

for line in text:

print(line)

class ErrorHandler(Handler):

def handle(self, request):

print("Invalid request")

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

report = Report(ReportFormat.TEXT)

pdf\_handler = PDFHandler()

text\_handler = TextHandler()

pdf\_handler.nextHandler = text\_handler

text\_handler.nextHandler = ErrorHandler()

pdf\_handler.handle(report)

2.Template Method

Template Method is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm’s structure.

class MakeMeal:

def prepare(self): pass

def cook(self): pass

def eat(self): pass

def go(self):

self.prepare()

self.cook()

self.eat()

class MakePizza(MakeMeal):

def prepare(self):

print("Prepare Pizza")

def cook(self):

print("Cook Pizza")

def eat(self):

print("Eat Pizza")

class MakeTea(MakeMeal):

def prepare(self):

print("Prepare Tea")

def cook(self):

print("Cook Tea")

def eat(self):

print("Eat Tea")

makePizza = MakePizza()

makePizza.go()

print(5\*"+")

makeTea = MakeTea()

makeTea.go()

output :

Prepare Pizza

Cook Pizza

Eat Pizza

+++++

Prepare Tea

Cook Tea

Eat Tea