Python Design Patterns

1. Creational Design Patterns

2. Structural Design Patterns

3. Behavioral Design Patterns

**1)** **Creational Design Patterns**

How objects are created.

**1) Singleton Method**

* It is a way to provide one and only one object of a particular type.
* It involves only one class to create methods and specify the objects.

class Borg:

# state shared by each instance

\_\_shared\_state = dict()

# constructor method

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

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

self.state = 'vehicles'

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

return self.state

# main method

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

veh1 = Borg() # object of class Borg

veh2 = Borg() # object of class Borg

veh3 = Borg() # object of class Borg

veh1.state = 'car'

veh2.state = 'bus'

print(veh1)

print(veh2)

veh3.state = 'jeep'

print(veh1)

print(veh2)

print(veh3)

output : bus

bus

jeep

jeep

jeep

**2) Factory Method**

* It provides one of the best ways to create an object.Objects are created without exposing the logic to client
* When a user calls a method such that we pass in a string and the return value as a new object is implemented through factory method

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()

Output :

<img></img>

<input></input>

<obj></obj>

**3)Builder Method**

* Builder Pattern is a unique design pattern which helps in building complex object using simple objects and uses an algorithmic approach.
* Builder class builds the final object in step-by-step procedure. This builder is independent of other objects.

class Director:

\_\_builder = None

def setBuilder(self, builder):

self.\_\_builder = builder

def getCar(self):

car = Car()

# First goes the body

body = self.\_\_builder.getBody()

car.setBody(body)

# Then engine

engine = self.\_\_builder.getEngine()

car.setEngine(engine)

# And four wheels

i = 0

while i < 4:

wheel = self.\_\_builder.getWheel()

car.attachWheel(wheel)

i += 1

return car

# The whole product

class Car:

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

self.\_\_wheels = list()

self.\_\_engine = None

self.\_\_body = None

def setBody(self, body):

self.\_\_body = body

def attachWheel(self, wheel):

self.\_\_wheels.append(wheel)

def setEngine(self, engine):

self.\_\_engine = engine

def specification(self):

print "body: %s" % self.\_\_body.shape

print "engine horsepower: %d" % self.\_\_engine.horsepower

print "tire size: %d\'" % self.\_\_wheels[0].size

class Builder:

def getWheel(self): pass

def getEngine(self): pass

def getBody(self): pass

class JeepBuilder(Builder):

def getWheel(self):

wheel = Wheel()

wheel.size = 22

return wheel

def getEngine(self):

engine = Engine()

engine.horsepower = 400

return engine

def getBody(self):

body = Body()

body.shape = "SUV"

return body

# Car parts

class Wheel:

size = None

class Engine:

horsepower = None

class Body:

shape = None

def main():

jeepBuilder = JeepBuilder() # initializing the class

director = Director()

# Build Jeep

print "Jeep"

director.setBuilder(jeepBuilder)

jeep = director.getCar()

jeep.specification()

print ""

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

main()

Ootput :

Jeep

body: SUV

engine horsepower: 400

tire size: 22

**2)Structural Design Patterns**

Simplify the structure by identifying relationship between objects.

**1)Decorator Method**

* Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
* The motive of a decorator pattern is to attach additional responsibilities of an object dynamically.

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("muhammad harshad")

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

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

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

Output : before : muhammad harshad

after : <i><u><b>muhammad harshad</b></u></i>

**2)Facade Method**

It provides an easier way to access methods of the underlying systems by providing a single entry point.By through facade class we can access all subsystem classes

class Cook(object):

'''

Facade class

Desc: Provides easy interface to prepare dish instead of calling three

different classes and making difficult for client to use.

'''

def prepareDish(self):

self.cutter = Cutter()

self.cutter.cutVegetables()

self.boiler = Boiler()

self.boiler.boilVegetables()

self.frier = Frier()

self.frier.fry()

class Cutter(object):

'''

System class

Desc: Cutter class provide feature of cutting vegetables

'''

def cutVegetables(self):

print("All vegetables are cut")

class Boiler(object):

'''

System class

Desc: Cutter class provide feature of boiling vegetables

'''

def boilVegetables(self):

print("All vegetables are boiled")

class Frier(object):

'''

System class

Desc: Cutter class provide feature of frying vegetables

'''

def fry(self):

print("All vegetables is mixed and fried.")

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

# Using facade class to prepare dish

cook = Cook()

cook.prepareDish()

output :All vegetables are cut

All vegetables are boiled

All vegetables is mixed and fried.

**3)Proxy Method**

Allows you to provide the replacement for an another object.A real-world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash and provides a means of accessing that cash when required.

class College:

'''Resource-intensive object'''

def studyingInCollege(self):

print("Studying In College....")

class CollegeProxy:

'''Relatively less resource-intensive proxy acting as middleman.

Instantiates a College object only if there is no fee due.'''

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

self.feeBalance = 1000

self.college = None

def studyingInCollege(self):

print("Proxy in action. Checking to see if the balance of student is clear or not...")

if self.feeBalance <= 500:

# If the balance is less than 500, let him study.

college = College()

college.studyingInCollege()

else:

# Otherwise, don't instantiate the college object.

print("Your fee balance is greater than 500, first pay the fee")

"""main method"""

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

# Instantiate the Proxy

collegeProxy = CollegeProxy()

# Client attempting to study in the college at the default balance of 1000.

# Logically, since he / she cannot study with such balance,

# there is no need to make the college object.

collegeProxy.studyingInCollege()

# Altering the balance of the student

collegeProxy.feeBalance = 100

# Client attempting to study in college at the balance of 100. Should succeed.

collegeProxy.studyingInCollege()

output:

Proxy in action. Checking to see if the balance of student is clear or not...

Your fee balance is greater than 500, first pay the fee

Proxy in action. Checking to see if the balance of student is clear or not...

Studying In College....

**Behavioral Design Patterns**

**1)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

**2) Chain of Responsibility**

* Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
* Chain the receiving objects and pass the request along the chain until an object handles it.

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)

output:

\*\*\*\*\*Monthly report\*\*\*\*\*

Things are going

really, really well.