**Investigación # 4**

**Patrones de diseño**

**Nombre: Yeselka Tyrrell Grupo #1 Lunes 7:00 – 10:00 pm**

**Patrones de diseño**

Un patrón de diseño es la forma reutilizable de una solución a un problema de diseño. La idea fue introducida por el arquitecto Christopher Alexander y ha sido adaptado para diversas otras disciplinas, especialmente la informática.

Una colección organizada de los patrones de diseño que se refieren a un campo particular se llama un lenguaje de patrones. Este lenguaje proporciona una terminología común para discutir las situaciones a las los diseñadores se enfrentan.

1. **Observer pattern**

El patrón de observador ( un subconjunto del modelo de publicación / suscripción ) es un patrón de diseño de software en el que un objeto , denominado objeto , mantiene una lista de sus dependientes , llamados observadores , y les notifica automáticamente cualquier cambio de estado , normalmente llamando a uno de sus métodos. Se utiliza principalmente para implementar sistemas de manejo de eventos distribuidos”.

**class** **Observable** :

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

self . \_\_observers = []

**def** register\_observer ( self , observer ):

self . \_\_observers . append ( observer )

**def** notify\_observers ( self , \* args , \*\* kwargs ):

**for** observer **in** self . \_\_observers :

observer . notify ( self , \* args , \*\* kwargs )

**class** **Observer** :

**def** \_\_init\_\_ ( self , observable ):

observable . register\_observer ( self )

**def** notify ( self , observable , \* args , \*\* kwargs ):

**print** ( 'Got' , args , kwargs , 'From' , observable )

subject = Observable ()

observer = Observer ( subject )

subject . notify\_observers ( 'test' )

1. **Decorator pattern**

El patrón Decorator responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

**def** establecer\_costo\_decorator(funcion):

**def** envoltorio1(instancia, costo):

funcion(instancia, costo)

**return** envoltorio1

**def** obtener\_costo\_decorator(costo\_adicional):

**def** envoltorio1(funcion):

**def** envoltorio2(instancia):

**return** funcion(instancia) + costo\_adicional

**return** envoltorio2

**return** envoltorio1

**class** **Cafe**(object):

@establecer\_costo\_decorator

**def** establecer\_costo(self, costo):

self.costo = costo

@obtener\_costo\_decorator(0.5)

@obtener\_costo\_decorator(0.7)

@obtener\_costo\_decorator(0.2)

**def** obtener\_costo(self):

**return** self.costo

cafe = Cafe()

cafe.establecer\_costo(1.0)

**print** cafe.obtener\_costo() *# 2.4*

1. **Command pattern**

En la programación orientada a objetos, el patrón de comando es un patrón de diseño en el que se utiliza un objeto para representar y encapsular toda la información necesaria para llamar a un método en un momento posterior. Esta información incluye el nombre del método, el objeto que posee el método y los valores de los parámetros del método.

class Switch:

""" The INVOKER class"""

def \_\_init\_\_(**self**, flipUpCmd, flipDownCmd):

**self**.\_\_flipUpCommand = flipUpCmd

**self**.\_\_flipDownCommand = flipDownCmd

def flipUp(**self**):

**self**.\_\_flipUpCommand.execute()

def flipDown(**self**):

**self**.\_\_flipDownCommand.execute()

class Light:

"""The RECEIVER Class"""

def turnOn(**self**):

print "The light is on"

def turnOff(**self**):

print "The light is off"

class Command:

"""The Command Abstract class"""

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

pass

*#Make changes*

def execute(**self**):

*#OVERRIDE*

pass

class FlipUpCommand(Command):

"""The Command class for turning on the light"""

def \_\_init\_\_(**self**,light):

**self**.\_\_light = light

def execute(**self**):

**self**.\_\_light.turnOn()

class FlipDownCommand(Command):

"""The Command class for turning off the light"""

def \_\_init\_\_(**self**,light):

Command.\_\_init\_\_(**self**)

**self**.\_\_light = light

def execute(**self**):

**self**.\_\_light.turnOff()

class LightSwitch:

""" The Client Class"""

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

**self**.\_\_lamp = Light()

**self**.\_\_switchUp = FlipUpCommand(**self**.\_\_lamp)

**self**.\_\_switchDown = FlipDownCommand(**self**.\_\_lamp)

**self**.\_\_switch = Switch(**self**.\_\_switchUp,**self**.\_\_switchDown)

def switch(**self**,cmd):

cmd = cmd.strip().upper()

try:

if cmd == "ON":

**self**.\_\_switch.flipUp()

elif cmd == "OFF":

**self**.\_\_switch.flipDown()

else:

print "Argument **\"**ON**\"** or **\"**OFF**\"** is required."

except **Exception**, msg:

print "Exception occured: %s" % msg

*# Execute if this file is run as a script and not imported as a module*

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

lightSwitch = LightSwitch()

print "Switch ON test."

lightSwitch.switch("ON")

print "Switch OFF test"

lightSwitch.switch("OFF")

print "Invalid Command test"

lightSwitch.switch("\*\*\*\*")

1. **Mediator pattern**

El patrón de mediador proporciona una interfaz unificada a un conjunto de interfaces en un subsistema. Este patrón se considera que es un modelo de comportamiento debido a la forma en que puede alterar los programas que se ejecutan comportamiento.

La idea es que habría una clase Mediador que es consciente de la funcionalidad de todas las clases en el sistema. Las clases son conscientes de su funcionalidad e interactuar con la clase Mediador. Siempre que hay una necesidad de interacción entre las clases, una clase envía información al Mediador y es responsabilidad del mediador de pasar esta información a la clase requerida. Por lo tanto la complejidad que ocurre debido a la interacción entre las clases se reduce.

import time

class TC:

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

self.\_tm = tm

self.\_bProblem = 0

def setup(self):

print("Setting up the Test")

time.sleep(1)

self.\_tm.prepareReporting()

def execute(self):

if not self.\_bProblem:

print("Executing the test")

time.sleep(1)

else:

print("Problem in setup. Test not executed.")

def tearDown(self):

if not self.\_bProblem:

print("Tearing down")

time.sleep(1)

self.\_tm.publishReport()

else:

print("Test not executed. No tear down required.")

def setTM(self, TM):

self.\_tm = tm

def setProblem(self, value):

self.\_bProblem = value

class Reporter:

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

self.\_tm = None

def prepare(self):

print("Reporter Class is preparing to report the results")

time.sleep(1)

def report(self):

print("Reporting the results of Test")

time.sleep(1)

def setTM(self, TM):

self.\_tm = tm

class DB:

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

self.\_tm = None

def insert(self):

print("Inserting the execution begin status in the Database")

time.sleep(1)

#Following code is to simulate a communication from DB to TC

import random

if random.randrange(1, 4) == 3:

return -1

def update(self):

print("Updating the test results in Database")

time.sleep(1)

def setTM(self, TM):

self.\_tm = tm

class TestManager:

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

self.\_reporter = None

self.\_db = None

self.\_tc = None

def prepareReporting(self):

rvalue = self.\_db.insert()

if rvalue == -1:

self.\_tc.setProblem(1)

self.\_reporter.prepare()

def setReporter(self, reporter):

self.\_reporter = reporter

def setDB(self, db):

self.\_db = db

def publishReport(self):

self.\_db.update()

rvalue = self.\_reporter.report()

def setTC(self, tc):

self.\_tc = tc

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

reporter = Reporter()

db = DB()

tm = TestManager()

tm.setReporter(reporter)

tm.setDB(db)

reporter.setTM(tm)

db.setTM(tm)

# For simplification we are looping on the same test.

# Practically, it could be about various unique test classes and their

# objects

while (True):

tc = TC()

tc.setTM(tm)

tm.setTC(tc)

tc.setup()

tc.execute()

tc.tearDown()

1. **Factory method pattern**

El patrón de fábrica es un patrón de diseño creacional utilizado en el desarrollo de software para encapsular los procesos que intervienen en la creación de objetos.

Patrón de la fábrica implica la creación de un súper clase que proporciona una interfaz abstracta para crear objetos de un tipo determinado, pero en lugar de tomar una decisión sobre qué objetos se hayan creado se aplaza esta decisión creación de sus subclases. Para apoyar esto hay una jerarquía de clases para la creación de los objetos que los intentos de la clase de fábrica de crear y devolver.

class Person:

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

self.name = None

self.gender = None

def getName(self):

return self.name

def getGender(self):

return self.gender

class Male(Person):

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

print "Hello Mr." + name class Female(Person):

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

print "Hello Miss." + name class Factory:

def getPerson(self, name, gender):

if gender == 'M':

return Male(name)

if gender == 'F':

return Female(name)

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

factory = Factory()

person = factory.getPerson("Chetan", "M")

1. **Facade pattern**

El patrón de la fachada es un patrón de diseño de ingeniería de software utilizada con la programación orientada a objetos. (El nombre es por analogía con una fachada arquitectónica.). Una fachada es un objeto que proporciona una interfaz simplificada a un gran cuerpo de código, tal como una biblioteca de clases. Fachada no es más que una interfaz que oculta los detalles en el interior y las complejidades de un sistema y proporciona un end‖ -front simplificada al cliente. Con el patrón de fachada, el cliente puede trabajar con la interfaz fácil y hacer el trabajo sin estar preocupado de las complejas operaciones que se realizan por el sistema.

import time

SLEEP = 0.5

# Complex Parts

class TC1:

def run(self):

print("###### In Test 1 ######")

time.sleep(SLEEP)

print("Setting up")

time.sleep(SLEEP)

print("Running test")

time.sleep(SLEEP)

print("Tearing down")

time.sleep(SLEEP)

print("Test Finished\n")

class TC2:

def run(self):

print("###### In Test 2 ######")

time.sleep(SLEEP)

print("Setting up")

time.sleep(SLEEP)

print("Running test")

time.sleep(SLEEP)

print("Tearing down")

time.sleep(SLEEP)

print("Test Finished\n")

class TC3:

def run(self):

print("###### In Test 3 ######")

time.sleep(SLEEP)

print("Setting up")

time.sleep(SLEEP)

print("Running test")

time.sleep(SLEEP)

print("Tearing down")

time.sleep(SLEEP)

print("Test Finished\n")

# Facade

class TestRunner:

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

self.tc1 = TC1()

self.tc2 = TC2()

self.tc3 = TC3()

self.tests = [i for i in (self.tc1, self.tc2, self.tc3)]

def runAll(self):

[i.run() for i in self.tests]

# Client

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

testrunner = TestRunner()

testrunner.runAll()

1. Proxy pattern

Un proxy, en su forma más general, es un funcionamiento de clase como una interfaz a otra cosa. El proxy podría interactuar con cualquier cosa: una conexión de red, un objeto grande en la memoria, un archivo, o algún otro recurso que es costoso o imposible de duplicar. Un ejemplo bien conocido del patrón de proxy es un objeto de referencia de puntero de recuento.

patrón proxy tiene tres elementos esenciales:

* sujeto real (que realiza los objetivos de la empresa, representada por el proxy).
* Clase Proxy (que actúa como una interfaz para las peticiones del usuario y protege el verdadero sujeto)
* cliente (que hace las peticiones para la realización del trabajo

import time

class SalesManager:

def work(self):

print("Sales Manager working...")

def talk(self):

print("Sales Manager ready to talk")

class Proxy:

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

self.busy = 'No'

self.sales = None

def work(self):

print("Proxy checking for Sales Manager availability")

if self.busy == 'No':

self.sales = SalesManager()

time.sleep(2)

self.sales.talk()

else:

time.sleep(2)

print("Sales Manager is busy")

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

p = Proxy()

p.work()

p.busy = 'Yes'

p.work()

1. Null pattern

La intención de un objeto nulo es encapsular la ausencia de un objeto, proporcionando una alternativa sustituible que ofrece por defecto adecuado hacer nada comportamiento. En resumen, un diseño en el que "no saldrá nada de nada"

Utilice el patrón de objeto nulo cuando

* un objeto requiere un colaborador. El patrón de objeto nulo no introduce esta colaboración - que hace uso de una colaboración que ya existe
* algunos casos colaborador no deben hacer nada
* desea resumen el manejo de null distancia desde el cliente

class Null:

def \_\_init\_\_(self, \*args, \*\*kwargs):

"""Ignore parameters."""

return None

def \_\_call\_\_(self, \*args, \*\*kwargs):

"""Ignore method calls."""

return self

def \_\_getattr\_\_(self, mname):

"""Ignore attribute requests."""

return self

def \_\_setattr\_\_(self, name, value):

"""Ignore attribute setting."""

return self

def \_\_delattr\_\_(self, name):

"""Ignore deleting attributes."""

return self

def \_\_repr\_\_(self):

"""Return a string representation."""

return "<Null>"

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

"""Convert to a string and return it."""

return "Null"

def test():

"""Perform some decent tests, or rather: demos."""

# constructing and calling

n = Null()

print(n)

n = Null('value')

print(n)

n = Null('value', param='value')

print(n)

n()

n('value')

n('value', param='value')

print(n)

# attribute handling

n.attr1

print('attr1', n.attr1)

n.attr1.attr2

n.method1()

n.method1().method2()

n.method('value')

n.method(param='value')

n.method('value', param='value')

n.attr1.method1()

n.method1().attr1

n.attr1 = 'value'

n.attr1.attr2 = 'value'

del n.attr1

del n.attr1.attr2.attr3

# representation and conversion to a string

assert repr(n) == '<Null>'

assert str(n) == 'Null'

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

test()

1. **strategy pattern**

El patrón Estrategia (Strategy) es un patrón de diseño para el desarrollo de software. Se clasifica como patrón de comportamiento porque determina cómo se debe realizar el intercambio de mensajes entre diferentes objetos para resolver una tarea. El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

**class** **Button**:

*"""A very basic button widget."""*

**def** \_\_init\_\_(self, submit\_func, label):

self.on\_submit = submit\_func *# Poner la funcion estrategia directamente*

self.label = label

*# Se crean dos objetos con estrategias diferentes*

button1 = Button(sum, "Add 'em")

button2 = Button(**lambda** nums: " ".join(map(str, nums)), "Join 'em")

*# Probamos cada boton*

numbers = range(1, 10) *# Lista del 1 al 9*

**print** button1.on\_submit(numbers) *# muestra "45"*

**print** button2.on\_submit(numbers) *# muestra "1 2 3 4 5 6 7 8 9"*

1. **Visitor pattern**

En la programación orientada a objetos y software de ingeniería, el visitante patrón de diseño es una manera de separar un algoritmo a partir de una estructura de objeto en el que opera. Un resultado práctico de esta separación es la posibilidad de añadir nuevas operaciones a estructuras de objetos existentes sin modificar esas estructuras. Es una manera de seguir el principio abierto / cerrado.

class Node(object):

pass

class A(Node):

pass

class B(Node):

pass

class C(A, B):

pass

class Visitor(object):

def visit(self, node, \*args, \*\*kwargs):

meth = None

for cls in node.\_\_class\_\_.\_\_mro\_\_:

meth\_name = 'visit\_'+cls.\_\_name\_\_

meth = getattr(self, meth\_name, None)

if meth:

break

if not meth:

meth = self.generic\_visit

return meth(node, \*args, \*\*kwargs)

def generic\_visit(self, node, \*args, \*\*kwargs):

print('generic\_visit '+node.\_\_class\_\_.\_\_name\_\_)

def visit\_B(self, node, \*args, \*\*kwargs):

print('visit\_B '+node.\_\_class\_\_.\_\_name\_\_)

a = A()

b = B()

c = C()

visitor = Visitor()

visitor.visit(a)

visitor.visit(b)

visitor.visit(c)