**DECORATORI DI FUNZIONI:**

È un modello di progettazione *strutturale* che consente di associare nuovi comportamenti agli oggetti, posizionandoli all'interno di oggetti ***wrapper*** speciali che contengono comportamenti.

I ***decoratori di funzione*** prendono come argomento una funzione per poi restituirne una funzione ***wrapper***, è un involucro contenente codice aggiuntivo che verrà eseguito prima dell’esecuzione della funzione originale.

**def** double(func):  
 @functools.wraps(func)  
 **def** wrapper(\*args, \*\*kwargs):  
 newList = list()  
 **for** item **in** args:  
 item \*= 2  
 newList.append(item)  
 args = newList  
 **return** func(\*args, \*\*kwargs)  
 **return** wrapper  
  
@double  
**def** add(a, b):  
 **return** a+b  
  
*#MAIN-------------------------------*print(add(2,3)) 🡪 10  
print(add.\_\_name\_\_) 🡪 add  
print(add.\_\_doc\_\_) 🡪 **None**

***wrapper*** accetta gli argomenti della funzione ***add (\*args, \*\*kwargs)*** e li moltiplica per 2.

Dal ***wrapper*** viene ritornata la funzione decorata coi parametri modificati, mentre dal decoratore viene ritornata la funzione wrapper, che verrà eseguita prima di add.

La funzione decorata avrà il valore dell’attributo ***\_\_name\_\_*** settato a “*wrapper*” invece che con il nome originale e non ha una ***docstring***, anche se viene esplicitata. Per ovviare a questo problema, si usa il decoratore ***@functools.wraps*** che può essere usato per decorare il ***wrapper*** dentro il decoratore, e assicurare che ***\_\_name\_\_*** e ***\_\_doc\_\_*** contengano i valori della funzione originale.

**DECORATORI DI CLASSE:**

I ***decoratori di classe*** prendono come argomento una classe, permettendo modifiche in ogni aspetto e funzione.

**def** dec\_counterClass(decoratedClass):  
 decoratedClass.numberOfInstances = 0  
 decoratedClass.oldInit = decoratedClass.\_\_init\_\_  
  
 **def** moddedInit(self, \*args, \*\*kwargs):  
 decoratedClass.numberOfInstances += 1  
 decoratedClass.oldInit(self, \*args, \*\*kwargs)  
  
 decoratedClass.\_\_init\_\_ = moddedInit  
 **return** decoratedClass  
  
@dec\_counterClass  
**class** Counter:  
 **pass***#MAIN------------------------------------------------*a = Counter()  
b = Counter()  
c = Counter()  
print(Counter.numberOfInstances) 🡪 3

Vogliamo fare in modo che la classe, alla creazione di una propria istanza, incrementi un contatore, tenendo conto di tutte le istanze della classe.

Si stabilisce una nuova funzione ***init*** (ovvero ***moddedInit***) che incrementi il contatore e che esegua il vecchio init, salvato in una variabile (***oldInit***), successivamente tale ***moddedInit*** viene salvato nella variabile init originaria. Infine viene ritornata la classe modificata.

**PROPERTY:**

Si può avere la necessità di variabili di istanza o classe che possano essere modificate solo con alcuni parametri o che possano essere utilizzate solo in alcune condizioni, limitandone l’accesso.

**class** MyClass:  
 **def** \_\_init\_\_(self):  
 self.\_\_var = 0  
  
 **def** getter(self):  
 print(**"GETTER: "**)  
 **return** self.\_\_var  
  
 **def** setter(self, value):  
 print(**"SETTER: "**, value)  
 **if** value > 0:  
 self.\_\_var = value  
  
 **def** deleter(self):  
 print(**"DELETER: "**, self)  
 **del** self.\_\_var  
  
 variabile = property(getter, setter, deleter, **"Doc"**)  
  
*#MAIN------------------------------------------------------*var = MyClass()  
var.variabile = 10 🡪 SETTER: 10  
print(var.variabile) 🡪 GETTER: 10  
**del** var.variabile 🡪 DELETER: <MyClass>

Grazie alla funzione ***property***, una variabile con lo stesso nome della variabile privata (*senza underscore*) potrà esser utilizzata come una normale variabile ma facendo riferimento alla privata, utilizzando le associazioni al posto delle funzioni.

**PRPERTY COI DECORATORI:**

**class** MyClass:  
 \_\_variabile = 0  
  
 @property  
 **def** variabile(self):  
 print(**"GETTER: "**)  
 **return** self.\_\_variabile  
  
 @variabile.setter  
 **def** variabile(self, value):  
 print(**"SETTER: "**, value)  
 **if** value > 0:  
 self.\_\_variabile = value  
  
 @variabile.deleter  
 **def** variabile(self):  
 print(**"DELETER: "**)  
 **del** self.\_\_variabile  
  
*#MAIN----------------------------------------*var = MyClass()  
var.variabile = 10 🡪 SETTER: 10  
print(var.variabile) 🡪 GETTER: 10  
**del** var.variabile 🡪 DELETER: <MyClass>

Il getter avrà come descrittore @property, il setter variabile.setter e il deleter @variabile.deleter.

**ENSURE:**

Questo tipo di decoratore permette di aggiungere ***getter***, ***setter*** e ***deleter*** ad un qualsiasi numero di variabili di una classe, evitando di scrivere codice ridondante.

**def** validation (value):  
 **if not** isinstance(value, int):  
 **raise** ValueError(**"This value is bad"**)  
  
**def** ensure(variableName, validationFunction, documentString):  
 **def** decorator(ClassToEdit):  
 privateVar= **"\_\_"**+ variableName  
 setattr(ClassToEdit, privateVar, 0)  
 **def** getter(self):  
 **return** getattr(self, privateVar)  
 **def** setter(self, value):  
 validationFunction(value)  
 setattr(self, privateVar, value)  
  
 setattr(ClassToEdit, variableName, property(getter, setter, documentString))  
 **return** ClassToEdit  
 **return** decorator  
  
@ensure(**"myVariable"**, validation, **"Document"**)  
**class** MyClass:  
 **pass***#Main-------------------------------------------------------------------------------*myVar= MyClass()  
myVar.myVariable= 10  
print(myVar.myVariable) 🡪 10

**SINGLETON:**

È un modello di progettazione *creazionale* che consente di garantire che una classe abbia solo un'istanza, fornendo al contempo un punto di accesso globale a questa istanza.

**class** Singleton:  
 \_\_instance = **None  
  
 def** \_\_init\_\_(self, Class, \*params):  
 **if** Singleton.\_\_instance **is None**:  
 Singleton.\_\_instance = Class(\*params)  
  
 **def** \_\_getattr\_\_(self, attr):  
 **return** getattr(self.\_\_instance, attr)  
  
 **def** \_\_setattr\_\_(self, attr, newValue):  
 **return** setattr(self.\_\_instance, attr, newValue)  
  
 **def** getid(self):  
 **return** id(self.\_\_instance)  
  
**class** MyClass:  
 **pass***#Main--------------------------------------------------*s1 = Singleton(MyClass)  
s1.var = 10  
print(**"ID : "**, s1.getid()) 🡪 ID : 14561712  
print(**"VAR: "**, s1.var) 🡪 VAR: 10  
s2 = Singleton(MyClass)  
print(**"ID : "**, s2.getid()) 🡪 ID : 14561712  
print(**"VAR: "**, s2.var) 🡪 VAR: 10

Al momento della sua istanziazione, il singleton accetta come parametro una classe e dei parametri, i quali inizializzeranno la classe passata come parametro.

Il problema è che passando la classe dall’esterno, essa è comunque istanziabile più volte tramite il suo costruttore: il singleton deve impedire ciò, perciò la classe viene specificata privata proprio per evitare che essa venga istanziata molteplici volte.

**BORG:**

Consente di creare più istanze di classe, ma queste condividono tra loro lo stesso stato. Il concetto di Borg è proprio quello di cambiare il ***\_\_dict\_\_*** di default quando si istanzia l’oggetto con la ***\_\_new\_\_*** : si stabilisce un dizionario di classe che conterrà tutte le variabili di istanza, in modo tale che siano comuni a tutte le istanze della classe Borg.

Quando viene creata un’istanza di una classe, viene invocato prima ***\_\_new\_\_*** (che crea l’oggetto), accettando ***cls*** come primo parametro perché quando viene invocato di fatto l’istanza deve essere ancora creata, e poi ***\_\_init\_\_*** (che inizializza le variabili di istanza).

class Borg:  
 \_\_varDict= {}

def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 objToReturn = super().\_\_new\_\_(cls, \*args, \*\*kwargs)  
 objToReturn.\_\_dict\_\_= cls.\_\_varDict  
 return objToReturn  
  
#MAIN------------------------------------------------------  
first = Borg()  
second = Borg()  
third = Borg()  
first.myVar = 10  
print(second.myVar) **🡪 10**  
second.anotherVar = 20  
print(third.anotherVar) **🡪 20**

***new*** crea una nuova istanza di ***cls*** invocando il metodo ***\_\_new\_\_*** della superclasse, che modifica l’istanza appena creata.

Se ***new*** restituisce un’istanza di ***cls*** allora viene invocata la ***init*** con gli stessi args.

**ADAPTER:**

È un modello di progettazione *strutturale* che consente la collaborazione di oggetti con interfacce incompatibili, permette ad una classe di adattarsi ad un’altra tramite una semplice interfaccia, senza cambiare il proprio codice.

class Computer:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def execute(self):  
 return f"The {self.name} Computer is executing"  
  
class Synthesizer:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def play(self):  
 return f"The {self.name} Synthesizer is playing"  
  
class Human:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def speak(self):  
 return f"The {self.name} Human is speaking"  
  
#Adattatore------------------------------------------------------  
class Adapter:  
 def \_\_init\_\_(self, obj, dictMethods):  
 self.obj = obj  
 self.\_\_dict\_\_.update(dictMethods)  
  
#MAIN------------------------------------------------------------  
pc = Computer("MyPC")  
synth = Synthesizer("MySynth")  
human = Human("MyHuman")  
  
lista = [pc]  
lista.append(Adapter(synth, dict(execute=synth.play)))  
lista.append(Adapter(human, dict(execute=human.speak)))  
  
for item in lista: **The MyPC Computer is executing**  
 print(item.execute()) **🡪 The MySynth Synthesizer is playing**

**The MyHuman Human is speaking**

L’adapter prende come argomenti un oggetto e un dizionario.

L’oggetto viene conservato nella variabile di istanza, mentre il dizionario è una coppia ***attributo***-***metodo***, in modo da salvare nel dizionario degli attributi, ovvero un attributo ***execute*** che contenga il metodo da richiamare.

**PROXY:**

È un modello di progettazione *strutturale* che consente di fornire un sostituto. Un proxy controlla l'accesso all'oggetto originale, consentendo di eseguire qualcosa prima o dopo che la richiesta arriva all'oggetto originale.

class MyClass:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def hello(self):  
 print("Hello I am:", self.name)  
 def goodbye(self):  
 print("Goodbye from: ", self.name)  
  
class GenericProxy:  
 def \_\_init\_\_(self, obj):  
 self.\_\_internalObj = obj  
 def \_\_getattr\_\_(self, attrName):  
 print("CONTROLPROXY")  
 return getattr(self.\_\_internalObj, attrName, None)  
  
#Main-----------------------------------------------------  
proxy = GenericProxy(MyClass("Mario"))  
proxy.hello() **🡪** **CONTROLPROXY Hello I am: Mario**  
proxy.goodbye() **🡪 CONTROLPROXY Goodbye from: Mario**

Il proxy fa da interfaccia completa all’oggetto che effettuerà le operazioni, implementa tutte le funzioni dell’oggetto interno.

Ovviamente il proxy può non implementare tutte le funzioni dell’oggetto interno, in modo da limitare le operazioni eseguibili dall’esterno

**CHAIN OF RESPONSBILITY:**

È un modello di progettazione *comportamentale* che consente di passare le richieste lungo una catena di gestori. Alla ricezione di una richiesta, ciascun gestore decide di elaborare la richiesta o di passarla al gestore successivo nella catena.

def coroutine(func):  
 @functools.wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 generator = func(\*args, \*\*kwargs)  
 next(generator)  
 return generator  
 return wrapper  
  
@coroutine  
def KeyboardHandler(successor=None):  
 while True:  
 event = (yield)  
 print("Invocazione KEYBOARD")  
 if event == "Keyboard":  
 print("Keyword is pressed")  
 elif successor is not None:  
 successor.send(event)  
  
@coroutine  
def MouseHandler(successor=None):  
 while True:  
 event = (yield)  
 print("Invocazione MOUSE")  
 if event == "Mouse":  
 print("Mouse is pressed")  
 elif successor is not None:  
 successor.send(event)  
  
# MAIN----------------------------------------------------------  
chain = MouseHandler(KeyboardHandler(None)) **Invocazione MOUSE**  
chain.send("Mouse") **🡪 Mouse is pressed**  
print("\n")  
chain.send("Keyboard") **🡪 Invocazione MOUSE**

**Invocazione KEYBOARD**

**Keyword is pressed**

Un generatore è una funzione che ha una o più espressioni ***yield*** al posto dell’istruzione return.

Una coroutine usa anch’essa l’espressione ***yield***, ma con un differente comportamento: viene eseguito un loop infinito e si sospende ogni volta che si raggiunge uno ***yield***, in attesa di un valore da gestire con quest’ultima espressione.

Se alla coroutine viene passato un valore, quest’ultimo verrà gestito dalla ***yield*** e continuerà a ciclare, fino ad arrivare nuovamente alla ***yield*** che attenderà un nuovo valore.

I valori vengono inseriti nella coroutine tramite i metodi ***send()*** o ***throw()***.

**STATE SENSITIVE:**

È un modello di progettazione *comportamentale* che consente a un oggetto di modificarne il comportamento quando cambia il suo stato. Nello ***State Sensitive Pattern*** cambia il comportamento dei metodi in base allo stato assunto.

class Multiplexer:  
 OFF, ON = [0, 1]  
 def \_\_init\_\_(self):  
 self.state = Multiplexer.ON  
 def connect(self):  
 if self.state == Multiplexer.ON:  
 print("Multiplexer CONNECTED")  
 else:  
 print("Operation not performed - multiplexer off")  
 def disconnect(self):  
 if self.state == Multiplexer.ON:  
 print("Multiplexer DISCONNECTED")  
 else:  
 print("Operation not performed - multiplexer off")  
#MAIN-------------------------------------------------------------------  
multiplexer = Multiplexer()  
multiplexer.connect() **🡪 Multiplexer CONNECTED**   
multiplexer.disconnect() **🡪 Multiplexer DISCONNECTED**  
multiplexer.state = Multiplexer.OFF  
multiplexer.connect() **🡪 Operation not performed-multiplexer is off**

**STATE SPECIFIC:**

Nello ***State Specific Pattern*** vengono utilizzati metodi diversi in base allo stato assunto. Lo stato diventa una proprietà, avrà un metodo per ottenere e settare lo stato ed i metodi sono privati, da eseguire in base allo stato assunto.

class MultiplexerSpecific:  
 OFF, ON = [0, 1]  
 def \_\_init\_\_(self):  
 self.state = MultiplexerSpecific.ON #Chiama il setter  
  
 @property  
 def state(self):  
 if self.connect != self.\_\_active\_connect:  
 return MultiplexerSpecific.OFF  
 else:  
 return MultiplexerSpecific.ON  
 @state.setter  
 def state(self, newState):  
 if newState == MultiplexerSpecific.ON:  
 self.connect = self.\_\_active\_connect  
 self.disconnect = self.\_\_active\_disconnect  
 else:  
 self.connect = lambda \*args: None  
 self.disconnect = lambda \*args: None  
  
 def \_\_active\_connect(self):  
 print(f"Connect, current status: {self.print\_status()}")  
 def \_\_active\_disconnect(self):  
 print(f"Disconnect, current status: {self.print\_status()}")  
 def print\_status(self):  
 if self.state == MultiplexerSpecific.ON:  
 return MultiplexerSpecific.ON  
 else:  
 return MultiplexerSpecific.OFF

#MAIN--------------------------------------------------------------  
multiplexer = MultiplexerSpecific()  
multiplexer.connect() **🡪 Connect,current status:on**  
multiplexer.disconnect() **🡪 Disconnect,current status:on**  
multiplexer.state = MultiplexerSpecific.OFF  
multiplexer.connect() **🡪**

**MEDIATOR(convenzionale):**

È un modello di progettazione *comportamentale* che consente di ridurre le dipendenze caotiche tra gli oggetti. Il modello limita le comunicazioni dirette tra gli oggetti e li costringe a collaborare solo tramite un oggetto mediatore.

class Component1:  
 def \_\_init\_\_(self, mediator = None):  
 self.\_\_mediator = mediator  
 @property  
 def mediator(self):  
 return self.\_\_mediator  
 @mediator.setter  
 def mediator(self, mediator):  
 self.\_\_mediator = mediator  
 def method1(self):  
 print("Component1 esegue method1")  
 self.mediator.notify(self)  
 def response1(self):  
 print("Component1 ha risposto")  
  
class Component2():  
 def \_\_init\_\_(self, mediator = None):  
 self.\_\_mediator = mediator  
 @property  
 def mediator(self):  
 return self.\_\_mediator  
 @mediator.setter  
 def mediator(self, mediator):  
 self.\_\_mediator = mediator  
 def method2(self):  
 print("Component2 esegue method2")  
 self.mediator.notify(self)  
 def response2(self):  
 print("Component2 ha risposto")  
  
class Mediator():  
 def \_\_init\_\_(self, component1, component2):  
 self.\_\_component1 = component1  
 self.\_\_component2 = component2  
 self.\_\_component1.mediator = self  
 self.\_\_component2.mediator = self  
 def notify(self, sender):  
 if sender == self.\_\_component1:  
 print("Mediator reagisce a Component1 e inoltra a Component2")  
 self.\_\_component2.response2()  
 elif sender == self.\_\_component2:  
 print("Mediator reagisce a Component2 e inoltra a Component1")  
 self.\_\_component1.response1()  
  
#MAIN-------------------------------------------------------------------------------  
c1 = Component1() **Component1 esegue method1**  
c2 = Component2() **Mediator reagisce a Component1 e inoltra a Component2**  
mediator = Mediator(c1, c2) **Component2 ha risposto**  
 **🡪**   
c1.method1() **Component2 esegue method2**  
print("\n", end="") **Mediator reagisce a Component2 e inoltra a Component1**  
c2.method2() **Component1 ha risposto**

(1)

1. Mediator incapsula le relazioni tra i vari componenti, mantenendo i loro riferimenti. I componenti non devono essere a conoscenza di altri componenti. Se succede qualcosa di importante all'interno o verso un componente, deve solo informare il mediatore. Quando il mediatore riceve la notifica, può facilmente identificare il mittente, che potrebbe essere sufficiente per decidere quale componente deve essere attivato in cambio.

**MEDIATOR(coroutine):**

def coroutine(func):  
 @functools.wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 generator = func(\*args, \*\*kwargs)  
 next(generator)  
 return generator  
 return wrapper  
  
class Component1:  
 def \_\_init\_\_(self):  
 self.\_\_mediator = None  
 @property  
 def mediator(self):  
 return self.\_\_mediator  
 @mediator.setter  
 def mediator(self, mediator):  
 self.\_\_mediator = mediator  
 def method1(self):  
 print("Component1 esegue method1")  
 self.mediator.send("method1")  
 def response1(self):  
 print("Component1 ha risposto")  
  
class Component2:  
 #init, getter e setter uguali a Component1  
 def method2(self):  
 print("Component2 esegue method2")  
 self.mediator.send("method2")  
 def response2(self):  
 print("Component2 ha risposto")  
  
class MediatorSet:  
 def \_\_init\_\_(self, component1, component2):  
 self.\_\_component1 = component1  
 self.\_\_component2 = component2  
 self.mediator = self.component1Mediator(self.component2Mediator())  
 self.\_\_component1.mediator = self.mediator  
 self.\_\_component2.mediator = self.mediator  
 @coroutine  
 def component1Mediator(self, successor = None):  
 while True:  
 event = (yield)  
 print("component1Mediator interpellato")  
 if event == "method1":  
 self.\_\_component2.response2()  
 elif successor is not None:  
 successor.send(event)  
 @coroutine  
 def component2Mediator(self, successor=None):  
 while True:  
 event = (yield)  
 print("component2Mediator interpellato")  
 if event == "method2":  
 self.\_\_component1.response1()  
 elif successor is not None:  
 successor.send(event)  
  
#MAIN---------------------------------------------------------------------  
c1 = Component1() **Component1 esegue method1**  
c2 = Component2() **component1Mediator interpellato**  
mediator = MediatorSet(c1, c2) **Component2 ha risposto**  
c1.method1() **🡪**   
print("\n", end="") **Component2 esegue method2**  
c2.method2() **component1Mediator interpellato**

**component2Mediator interpellato**

**Component1 ha risposto**

Un mediatore può essere considerato come una pipeline che riceve messaggi per poi passarli agli oggetti interessati: ciò può esser realizzabile tramite le coroutine.

**TEMPLATE METHOD:**

È un modello di progettazione *comportamentale* che definisce lo scheletro di un algoritmo nella superclasse ma consente di definire un algoritmo affidandone l’esecuzione ad opportune sottoclassi.

class AbstractWordCouter:  
 @staticmethod  
 def canCount(filename):  
 raise NotImplementedError  
 @staticmethod  
 def count(filename):  
 raise NotImplementedError

class TextWordCounter(AbstractWordCouter):  
 @staticmethod  
 def canCount(filename):  
 return filename.lower().endswith(".txt")  
 @staticmethod  
 def count(filename):  
 if TextWordCounter.canCount(filename):  
 counter=0  
 with open(filename, "r") as f:  
 for line in f:  
 arrayOfWords = line.split()  
 print(arrayOfWords)  
 for word in arrayOfWords:  
 counter +=1  
 return counter  
  
#MAIN-------------------------------------------------------------  
print("There are", TextWordCounter.count("testFile.txt"), "words")

**🡪 ['Programming', 'with', 'Python', '!!!']**

**There are 4 words**

Ogni metodo definito nella superclasse viene dichiarato statico: questo perché non si vede l’utilità di salvare alcuno stato nelle istanze o nella classe.

Se la classe può effettuare il conteggio di parole sul file indicato, allora si esegue il conteggio e si restituisce il valore, se la classe non può effettuare il conteggio di parole, allora viene restituito 0.

**OBSERVER:**

È un modello di progettazione *comportamentale* che consente di definire un meccanismo di sottoscrizione per notificare a più oggetti eventuali eventi che si verificano sull'oggetto che stanno osservando.

class Observed:  
 def \_\_init\_\_(self):  
 self.\_\_observers = set()  
 def observers\_add(self, observer, \*observers):  
 for observer in chain((observer,), observers):  
 self.\_\_observers.add(observer)  
 observer.update(self)  
 def observer\_discard(self, observer):  
 self.\_\_observers.discard(observer)  
 def observers\_notify(self):  
 for observer in self.\_\_observers:  
 observer.update(self)  
  
#Oggetto Osservato:  
class SliderModel(Observed):  
 def \_\_init\_\_(self, minimum, value, maximum):  
 super().\_\_init\_\_()  
 self.\_\_minimum = self.\_\_value = self.\_\_maximum = None  
 self.minimum = minimum  
 self.value = value  
 self.maximum = maximum  
 @property  
 def value(self):  
 return self.\_\_value  
 @value.setter  
 def value(self, value):  
 if self.\_\_value != value:  
 self.\_\_value = value  
 self.observers\_notify()  
  
#Oggetto Osservatore:  
class HistoryView:  
 def \_\_init\_\_(self):  
 self.data = []  
 def update(self, model):  
 self.data.append((model.value, time()))  
  
#MAIN----------------------------------------------------------------  
historyView = HistoryView()  
  
model = SliderModel(0, 0, 40)  
model.observers\_add(historyView)  
  
for value in (7, 23, 37):  
 model.value = value  
  
for value, timestamp in historyView.data:  
 print("{:3} {}".format(value, datetime.fromtimestamp(timestamp)))

**🡪 0 2019-12-21 18:08:43.890876**

**7 2019-12-21 18:08:43.890876**

**23 2019-12-21 18:08:43.890876**

**37 2019-12-21 18:08:43.890876**

***Observer*** mantiene un insieme di oggetti osservatori ed è caratterizzata da funzioni capaci di gestire gli osservatori.

***SliderModel*** eredita ***Observer*** acquisendo un insieme di osservatori inizialmente vuoto ed i metodi per la gestione degli osservatori, implementati da ***Observer***. Ogni volta che il modello cambia il proprio valore, vengono notificate tutte le viste tramite il metodo ***notifyObserver***.

***HistoryView*** è un osservatore del modello, fornisce un metodo update che verrà richiamato ad ogni cambiamento di un qualsiasi modello collegato.

**FACADE:**

È un modello di progettazione *strutturale* che fornisce un'interfaccia semplificata a una libreria, un framework o qualsiasi altro insieme complesso di classi.

class SubSystem1:  
 def \_\_init\_\_(self, val):  
 self.value1 = val  
 def myFunction1(self):  
 print("Valore SubSystem1 : ", self.value1)  
  
class SubSystem2:  
 def \_\_init\_\_(self, val):  
 self.value2 = val  
 def myFunction2(self):  
 print("Valore SubSystem2 : ", self.value2)  
 def myFunction2double(self):  
 print("Valore SubSystem2 : ", self.value2\*2)  
  
class Facade:  
 def \_\_init\_\_(self, item1, item2):  
 self.\_\_subsystem1 = item1  
 self.\_\_subsystem2 = item2  
 def executeAll(self):  
 print("Facade inizializza i sottosistemi: ")  
 self.\_\_subsystem1.myFunction1()  
 self.\_\_subsystem2.myFunction2()  
 self.\_\_subsystem2.myFunction2double()  
  
#MAIN-----------------------------------------------------------  
sub1 = SubSystem1(100)  
sub2 = SubSystem2(200)  
myFacade = Facade(sub1, sub2)  
myFacade.executeAll() **🡪 Facade inizializza i sottosistemi**

**Valore SubSystem1 : 100**

**Valore SubSystem2 : 200**

**Valore SubSystem2 : 400**

Le classi dei sottosistemi non sono a conoscenza dell'esistenza della facciata. Operano all'interno del sistema e lavorano direttamente tra loro.

Il client utilizza la facciata invece di chiamare direttamente gli oggetti del sottosistema.

***Context Managers*:**

Permettono di allocare e rilasciare risorse in maniera decisamente semplificata. Esempio la keyword “***with***”, la quale apre un file, opera all’interno e lo chiude automaticamente.

myFile = open("testFile.txt", "w")  
myFile.write("Hello World?")  
myFile.close()

**=**

with MyFile("testFile.txt", "w") as myFile:  
 myFile.write("Hello World")

Una classe ***context manager*** deve forzatamente definire i metodi ***\_\_enter\_\_*** ed ***\_\_exit\_\_.***

***\_\_enter\_\_*** deve poter ritornare l’oggetto allocato, mentre ***\_\_exit\_\_*** deve poter liberare la memoria riservata all’oggetto allocato. L’allocazione dell’oggetto e la sua inizializzazione avvengono nell’init.

class MyFile:  
 def \_\_init\_\_(self, filename, method):  
 self.obj\_file = open(filename, method)  
 self.obj\_file.write("INIZIALIZZAZIONE \n")  
 def \_\_enter\_\_(self):  
 return self.obj\_file  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 if exc\_type is not None:  
 print("Exception not handled!")  
 self.obj\_file.close()  
 return False  
 else:  
 self.obj\_file.close()  
  
#MAIN---------------------------------------------  
with **MyFile**("testFile.txt", "w") as myFile:  
 myFile.write("Hello World")

1. Eseguita l’istruzione “***with MyFile(…)***”, viene aperto il blocco ***with*** e viene eseguita la ***\_\_init\_\_*** della classe ***MyFile***, la quale apre un file e salva il riferimento in una variabile di istanza.
2. Eseguita l’istruzione ***as myFile***, richiamando il metodo ***\_\_enter\_\_,*** che deve restituire la variabile di istanza, ***obj\_file***.
3. Eseguite le istruzioni interne al blocco ***with***.
4. Eseguito il metodo ***\_\_exit\_\_***, che libela la memoria allocata da ***\_\_init\_\_.***

**FLYWEIGHT:**

È un modello di progettazione *strutturale* che consente di adattare più oggetti alla quantità disponibile di RAM condividendo parti comuni di stato tra più oggetti invece di conservare tutti i dati in ciascun oggetto.

class Point:  
 \_\_slots\_\_ = ("x", "y", "z", "color")  
  
 def \_\_init\_\_(self, x,y,z,color = None):  
 self.x = x  
 self.y= y  
 self.z= z  
 self.color = color  
  
#MAIN--------------------------------------  
myPoint= Point(1,2,3,"Blue")  
print(myPoint.x) **🡪 1**  
print(myPoint.color) **🡪 Blue**

Conserva le variabili per la raffigurazione tridimensionale senza l’utilizzo del dict, in modo da risparmiare spazio in RAM. In tal modo nessun Point avrà il proprio dict privato. Tuttavia significa che non è possibile aggiungere attributi ai singoli punti, siccome la memoria sarà stata allocata staticamente.

Tale pattern serve a migliorare le prestazioni del programma, o meglio la memoria RAM richiesta per l’allocazione di determinati oggetti.

***La variabile \_\_slots\_\_*:**

Permette di risparmiare RAM: essendo il dict davvero pesante, è possibile sostituirlo con ***\_\_slots\_\_,*** il quale occupa circa il 40-50% di spazio in meno. Python usa un dict per conservare gli attributi di un’istanza, allocando memoria dinamica per ogni dict. Python non può allocare un quantitativo statico di memoria alla creazione dell’oggetto per poter conservare gli attributi, è possibile utilizzare ***\_\_slots\_\_*** per allocare spazio solo per quel determinato set di attributi.

È bene precisare che l’utilizzo di ***\_\_slots\_\_*** *esclude* l’utilizzo della variabile ***dict*** contenente le variabili di istanza: ***\_\_slots\_\_*** conterrà le variabili di istanza allocando staticamente memoria, dopo l’inizializzazione, non sarà possibile aggiungere nuove variabili di istanza.

**PROTOTYPE:**

È un modello di progettazione *creazionale* che consente di copiare oggetti esistenti senza che il codice dipenda dalle loro classi.

class Point:  
 \_\_slots\_\_= ("x", "y")  
 def \_\_init\_\_(self,x,y):  
 self.x = x  
 self.y = y  
  
#MAIN-------------------------------------------------------------------------  
original = Point(1,2)  
prototype = copy.deepcopy(original)  
prototype.x= 10  
prototype.y= 20  
print(f"ORIGINAL : x={original.x}, y={original.y}") **🡪 ORIGINAL : x=1, y=2**  
print(f"PROTOTYPE: x={prototype.x}, y={prototype.y}") **🡪 PROTOTYPE: x=10, y=20**