Programmazione Avanzata

Design Pattern: Decorator

Lezione VIII

Design Pattern

- Nel 1994, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides pubblicarono un libro intitolato **Design Patterns - Elements of Reusable Object-Oriented Software** in cui è stato introdotto il concetto di Design Pattern nell'Object Oriented Design (OOD).
- Il gruppo dei quattro autori è noto con il nome di **Gang of Four (GOF)**.
- Nel libro viene riportato il seguente pensiero dell'architetto Christopher Alexander: "Ciascun pattern descrive un problema che si presenta più e più volte nel nostro ambiente e poi descrive il nucleo della soluzione del problema, in modo tale che tu possa riutilizzare questa soluzione un milione di volte, senza mai applicarla alla stessa maniera."
- I quattro autori osservano che ciò che Christopher Alexander esprime riguardo ai pattern negli edifici e nelle città è vero anche quando si parla di object-oriented design pattern.
  - Solo che le soluzioni sono descritte in termini di interfacce e oggetti invece che di muri e porte.

Design Pattern

- Forniscono schemi generali per la soluzione di problematiche ricorrenti che si incontrano durante lo sviluppo del software
- Favoriscono il riutilizzo di tecniche di design di successo nello sviluppo di nuove soluzioni
- Evitano al progettista di riscoprire ogni volta le stesse cose
- Permettono di sviluppare un linguaggio comune che semplifica la comunicazione tra le persone coinvolte nello sviluppo del software

Design Pattern

- Per definire un design pattern occorre specificare:
- **Il nome del pattern.** Associare dei nomi ai design pattern consente un più elevato livello di astrazione nella fase di progettazione e facilita la comunicazione tra gli addetti ai lavori e la documentazione.
- **Il problema.** Il problema descrive in quali contesti ha senso applicare il pattern.
- **La soluzione.** La soluzione fornisce la descrizione astratta di un problema (nel nostro caso di OOD) e indica come utilizzare gli strumenti a disposizione (nel nostro caso classi e oggetti) per risolverlo.
- **Le conseguenze.** Le conseguenze descrivono i risultati dell'applicazione del design pattern. Esse sono fondamentali per valutare le diverse alternative e comprendere i costi e i benefici risultanti dall'applicazione del pattern

Design Pattern: elenco

- 1.Adapter
- 2.Facade
- 3.Composite
- 4.Decorator
- 5.Bridge
- 6.Singleton
- 7.Proxy
- 8.Flyweight
- 9.Strategy
- 10.State
- 11.Command
- 12.Observer

13. Memento
14.Interpreter
15.Iterator
16. Visitor
17.Mediator
18.Template Method
19.Chain of Responsibility
20.Builder
21.Prototype
22.Factory Method
23.Abstrac Factory

Design Pattern: classificazione

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Design Pattern: classificazione

- I pattern creazionali riguardano il processo di creazione degli oggetti
- I pattern strutturali riguardano la composizione di classi ed oggetti
- I pattern comportamentali caratterizzano i modi in cui le classi e gli oggetti interagiscono tra di loro e si distribuiscono le responsabilità

Design Pattern Creazionali

- I design pattern creazionali astraggono il processo di creazione
- Aiutano a rendere il sistema indipendente da come i suoi oggetti sono creati, composti e rappresentati
- I design pattern di questo tipo diventano sempre più utili man mano che il sistema diventa sempre più dipendente dalla composizione di oggetti. Man mano che ciò accade, l'enfasi si sposta dalla codifica di un insieme fissato di comportamenti alla definizione di un insieme più piccolo di comportamenti fondamentali che possono essere composti per dar vita a comportamenti più complessi

Design Pattern strutturali

- I Design Pattern strutturali riguardano le relazioni tra entità quali classi e oggetti
  - forniscono metodologie semplici per comporre oggetti per creare nuove funzionalità

Design Pattern Comportamentali

- I pattern comportamentali riguardano il modo in cui le cose vengono fatte, in altre parole, gli algoritmi e le interazioni tra oggetti.

- Forniscono modi efficaci per pensare e organizzare la computazione.
- Alcuni di questi pattern sono built-in in Python.

Riferimenti

- Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides **“Design Patterns - Elements of Reusable Object-Oriented Software”,** Addison-Wesley

- Mark Summerfield, **“Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns ”,** Addison-Wesley Professional

Il pattern Decorator

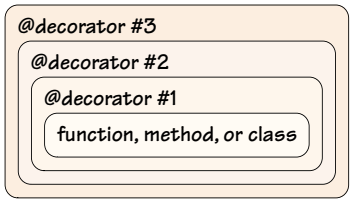
- È un design pattern **strutturale**.
- Serve quando vogliamo estendere le funzionalità di singoli oggetti dinamicamente
- Ad esempio un sistema GUI dovrebbe consentire di aggiungere proprietà (ad esempio, i bordi) o comportamenti (ad esempio, lo scrolling) ad ogni componente dell'interfaccia utente.
- Un modo per far questo è l'ereditarietà: ereditare un bordo da un'altra classe mette un bordo intorno ad ogni istanza della sottoclasse.
- Ciò è poco flessibile perché la scelta di un bordo è fatta in modo statico. Non è possibile controllare come e quando decorare la componente con un bordo.
- Un approccio più flessibile consiste nel racchiudere la componente in un altro oggetto che si occupa di aggiungere il bordo.
- Tale oggetto è chiamato decoratore.
- Il decoratore inoltra richieste alla componente e può svolgere azioni aggiuntive, come aggiungere un bordo o altre proprietà.

Il pattern Decorator

- Un *decoratore di funzione* è una funzione che ha come unico argomento una funzione e restituisce una funzione con lo stesso nome della funzione originale ma con ulteriori funzionalità
- Un *decoratore di classe* è una funzione che ha come unico argomento una classe e restituisce una classe con lo stesso nome della classe originale ma con funzionalità aggiuntive.
  - I decoratori di classe possono a volte essere utilizzati come alternativa alla creazione di sottoclassi
- In python c'è un supporto built-in per i decoratori di funzioni (e di metodi) e per i decoratori di classe.

Function Decorator

- Tutti i decoratori di funzioni o di metodi hanno la stessa struttura
  - Creazione della funzione wrapper:
    - All'interno del wrapper invochiamo la funzione originale.
    - Prima di invocare la funzione originale possiamo effettuare qualsiasi lavoro di preprocessing
    - Dopo la chiamata siamo liberi di acquisire il risultato, di fare qualsiasi lavoro di postprocessing e di restituire qualsiasi valore vogliamo.
  - Alla fine restituiamo la funzione wrapper come risultato del decoratore e questa funzione sostituisce la funzione originale acquisendo il suo nome.
  - Applicazione di un decoratore:
    - Si scrive il simbolo @, allo stesso livello di indentazione dello statement def seguito immediatamente dal nome del decoratore.
    - è possibile applicare un decoratore ad una funzione decorata.



Function Decorator

- La funzione mean() senza decoratore ha due o più argomenti numerici e restituisce la loro media come un float.
- Senza il decoratore la chiamata mean(5, "6", "7.5") genera un TypeError perché non è possibile sommare int e str.

```
@float_args_and_return
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
```

Function Decorator

- La funzione mean() decorata con il decoratore float_args_and_return può accettare due o più argomenti di qualsiasi tipo che convertirà in un float.
- Con la versione decorata, mean(5, "6", "7.5") non genera errore dal momento che float("6") and float("7.5") producono numeri validi.

.

```
def float_args_and_return(function):
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

Function Decorator

- Nel codice in basso, è stata creata la funzione senza il decoratore e poi sostituita con una nuova funzione invocando il decoratore
- A volte è necessario invocare i decoratori direttamente
  - vedremo in seguito degli esempi

```
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
mean = float_args_and_return(mean)
```

Function Decorator

- La funzione float_args_and_return() è un decoratore di funzione per cui ha come argomento una singola funzione
- Per convenzione, le funzioni wrapper hanno come argomenti un parametro che indica un numero variabile di parametri (*args, nell'esempio) e un parametro di tipo keyword (**kwargs, nell'esempio)
- Eventuali vincoli sugli argomenti sono gestiti dalla funzione originale. Nel creare il decoratore, dobbiamo solo assicurarci che alla funzione originale vengano passati tutti gli argomenti.

Function Decorator

- Per come è stato scritto il decoratore float_args_and_return ,
  - la funzione decorata avrà il valore dell'attributo __name__ settato a "wrapper" invece che con il nome originale della funzione
  - non ha docstring anche nel caso in cui la funzione originale abbia una docstring
- Per ovviare a questo inconveniente, la libreria standard di Python include il decoratore **@functools.wraps** che può essere usato per decorare una funzione wrapper dentro il decoratore e assicurare che gli attributi __name__ and __doc__ della funzione decorata contengano rispettivamente il nome e la docstring della funzione originale.

Function Decorator

- La versione in basso del decoratore float_args_and_return usa il decoratore @functools.wraps per garantire che la funzione wrapper()
  - abbia il suo attributo __name__ correttamente settato con il nome della funzione passata come argomento al decoratore (mean, nel nostro esempio)
  - abbia la docstring della funzione originale (non presente, nel nostro esempio)
- è sempre consigliabile usare il decoratore @functools.wraps dal momento che il suo uso ci assicura che
  - nei traceback vengano visualizzati i nomi corretti delle funzioni
  - si possa accedere alle docstring delle funzioni originali

```
def float_args_and_return(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

Function Decorator: esercizio

- Scrivere il decoratore di funzione decf che fa in modo che venga lanciata l'eccezione TypeError se il numero di argomenti è diverso da due. Altrimenti, se la funzione decorata restituisce un risultato, questo viene aggiunto insieme al valore del primo argomento in un file di nome "**risultato.txt**".
- Suggerimento: Ricordatevi di convertire a stringa il valore del primo argomento e il risultato quando li scrivete nel file e di aprire il file in modo da non cancellare quanto scritto precedentemente nel file.

Function Decorator: soluzione esercizio

```
from functools import wraps

def decf(f):
    @wraps(f)
    def wrapper(*args,**kwargs):
        if len(args)+len(kwargs)!=2:
            raise TypeError
        else:
            f_o=open("risultato.txt",'a')
            res=f(*args,**kwargs)
            if res!=None:
                f_o.write(res)
            if args:
                f_o.write(str(args[0]))
            else :
                f_o.write(str(next(iter(kwargs.values()))))
            f_o.write("\n")
            f_o.close()
    return wrapper
```

Function Decorator: esercizio

- Scrivere il decoratore di funzione **decora** che trasforma la funzione decorata in una funzione che lancia l'eccezione **TypeError** se uno o più argomenti non sono di tipo str. La funzione deve restituire una stringa formata dagli argomenti ricevuti in input e dal risultato intervallati da uno spazio. Non dimenticate di convertire il risultato in stringa quando lo inserite nella stringa output.
- Esempio: se la funzione riceve in input **"il" , "risultato", "è" ,** la funzione non lancia l'eccezione e restituisce la stringa "Il risultato è …" dove al posto dei puntini deve apparire il risultato della funzione.

Class Decorator

- I decoratori di classe sono simili ai decoratori di funzioni ma sono eseguiti al termine di uno statement class
- I decoratori di classe sono funzioni che ricevono una classe come unico argomento e restituiscono una nuova classe con lo stesso nome della classe originale ma con funzionalità aggiuntive.
  - I decoratori di classe possono essere usati sia per gestire le classi dopo che esse sono state create sia per inserire un livello di logica extra (wrapper) per gestire le istanze della classe quando sono create.

```
def decorator(aClass): ...

@decorator
class C: ...
```

è equivalente a

```
def decorator(aClass): ...

class C: ...
C = decorator(C)
```

Class Decorator

- è possibile usare questo decoratore per dotare automaticamente le classi con una variabile numInstances per contare le istanze.
- è possibile usare lo stesso approccio per aggiungere altri dati

classdec0.py

```
def count(aClass):
    aClass.numInstances = 0
    return aClass

@count
class Spam:
    def __init__(self):
        Spam.numInstances =Spam.numInstances + 1

class Sub(Spam):
    pass
class Other(Spam):
    pass
```

```
>>> from classdec0.py import Spam, Sub,
Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
3
>>> print(sub.numInstances)
3
>>> print(other.numInstances)
3
```

Class Decorator

- Per come è stato definito nella slide precedente, il decoratore count può essere applicato sia a classi che a funzioni

```
@count                          #equivalente a f=count(f)
def f(): pass

@count                          #equivalente a Other=count(Other)
class Other: pass


spam.numInstances               #entrambi settati a 0
Other.numInstances
```

Class Decorator

```
def count(aClass):
   aClass.numInstances = 0
   return aClass

@count
class Spam:
   def __init__(self):
      Spam.numInstances =Spam.numInstances + 1

@count
class Sub(Spam):
   pass

@count
class Other(Spam):
   def __init__(self):
      Other.numInstances =Other.numInstances + 1
```

classdec1.py

- In questo esempio ogni classe ha la sua variabile numInstances.
- Quando viene creato un oggetto di tipo Sub viene invocato __init__ della classe base Spam e viene incrementato numInstances di Spam
- Quando viene creato un oggetto di tipo Other viene invocato __init__ di Other e incrementato numInstances di Other

```
>>> from classdec1.py import Spam, Sub, Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
2
>>> print(sub.numInstances)
0
>>> print(other.numInstances)
1
```

Class Decorator

classdec2.py

```
def count(aClass):
    aClass.numInstances = 0
    return aClass

@count
class Spam:
    @classmethod
    def count(cls):
        cls.numInstances+=1

    def __init__(self):
        self.count()

@count
class Sub(Spam):
    pass

@count
class Other(Spam):
    pass
```

```
>>> from classdec2.py import Spam, Sub, Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
1
>>> print(sub.numInstances)
1
>>> print(other.numInstances)
1
>>> other=Other()
>>> print(other.numInstances)
2
>>> print(spam.numInstances)
1
>>> print(sub.numInstances)
1
```