Programmazione Avanzata

Design Pattern (I parte)

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 riusare 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

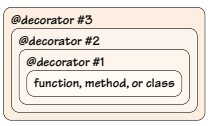
- è un design pattern **strutturale**.
- Serve quando vogliamo estendere le funzionalità di singoli oggetti dinamicamente
- Ad esempio un sistema GUI dovrebbe consentire di aggiungere proprietà (ad esempio, i bordi) o comportamenti (ad esempio, lo scrolling) ad ogni componente dell'interfaccia utente.
- Un modo per far questo è l'ereditarietà: ereditare un bordo da un'altra classe mette un bordo intorno ad ogni instanza della sottoclasse.
- Ciò è poco flessibile perche' 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 il decoratore 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 varibile 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 (vuota, 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
```

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 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 possible usare questo decoratore per dotare automaticamente le classi con un contatore di istanze.
- è possibile usare lo stesso approccio per aggiungere altri dati

```
def count(aClass):
        aClass.numInstances = 0
        return aClass
@count
class Spam: ...
@count
class Sub(Spam): ...
@count
class Other(Spam): ...
```

Class Decorator

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

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

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

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


Proprietà

- Per capire il prossimo esempio di class decorator occorre parlare degli attributi property
- La funzione built-in property permette di associare operazioni di fetch e set ad attributi specifici
- property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*) restituisce un attributo property
  - *fget* è una funzione per ottenere il valore di un attributo
  - *fset* è una funzione per settare un attributo
  - *fdel* è una funzione per cancellare un attributo
  - *doc* crea una docstring dell'attributo.

Proprietà

- Se c è un'istanza di C, c.x =value invocherà il setter setx e del c.x invocherà il deleter delx.
- Se fornita, doc sarà la docstring dell'attributo property. In caso contrario, viene copiata la docstring di *fget* (se esiste)

```
class C:
    def __init__(self):
        self._x = None
    def getx(self): return self._x
    def setx(self, value): self._x = value
    def delx(self): del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```


Proprietà

- Nella classe Parrot in basso usiamo il decoratore @property per trasformare il metodo voltage() in un "getter" per l'attributo **read-only** voltage e settare la docstring di voltage a "Get the current voltage."

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

Proprietà

- Un oggetto property ha i metodi getter, setter e deleter che possono essere usati come decoratori per creare una copia della proprietà con la corrispondente funzione accessoria uguale alla funzione decorata
- Questi due codici sono equivalenti
  - nel codice a sinistra dobbiamo stare attenti a dare alla funzioni aggiuntive lo stesso nome della proprietà originale (x, nel nostro esempio).

```
class C:
        def __init__(self):
                self._x = None
        @property
        def x(self):
                """I'm the 'x' property."""
                return self._x
        @x.setter
        def x(self, value):
                self._x = value
        @x.deleter
        def x(self):
                del self._x
```

```
class C:
        def __init__(self):
            self._x = None
        def getx(self): return self._x
        def setx(self, value): self._x = value
        def delx(self): del self._x

        x = property(getx, setx, delx, "I'm the 'x' property.")
```


Class Decorator

- È abbastanza comune creare classi che hanno molte proprietà read-write. Tali classi hanno molto codice duplicato o parzialmente duplicato per i getter e i setter.
- Esempio: Una classe Book che mantiene il titolo del libro, lo ISBN, il prezzo, e la quantità. Vorremmo
  - quattro decoratori @property, tutti fondamentalmente con lo stesso codice (ad esempio, @property def title(self): return title).
  - quattro metodi setter il cui codice differirebbe solo in parte
- I decoratori di classe consentono di evitare la duplicazione del codice

Class Decorator

```
@ensure("title", is_non_empty_str)
@ensure("isbn", is_valid_isbn)
@ensure("price", is_in_range(1, 10000))
@ensure("quantity", is_in_range(0, 1000000))
class Book:

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

self.title, self.isbn, self.price, self.quantity sono proprietà per cui gli assegnamenti che avvengono in __init__() sono tutti effettuati dai setter delle proprietà

Invece di scrivere il codice per creare le proprietà con i loro getter e setter, si usa un decoratore di classe

La funzione ensure() è un **decorator factory,** cioè una funzione che restituisce un decoratore. La funzione ensure() accetta due parametri, il nome di una proprietà e una funzione di validazione, e restituisce un decoratore di classe

Nel codice applico 4 volte @ensure per creare le 4 proprietà in questo ordine: quantity, price, isbn, title

Class Decorator

- Lo statement class Book deve essere eseguito per primo perché la classe Book serve come parametro di ensure("quantity",...)
- la classe ottenuta applicando il decoratore restituito da ensure("quantity",...) serve come parametro in ensure("price",...)e così via.

```
ensure("title", is_non_empty_str)( # Pseudo-code
    ensure("isbn", is_valid_isbn)(
        ensure("price", is_in_range(1, 10000))(
            ensure("quantity", is_in_range(0, 1000000))(class Book: ...))))
```

Class Decorator

- La funzione ensure() crea un decoratore di classe parametrizzato dal nome della proprietà (name), dalla funzione di validazione (validate) e da una docstring opzionale (doc).
- Ogni volta che un decoratore di classe restituito da ensure() è usato per una particolare classe, quella classe viene dotata della proprietà il cui nome è specificato dal primo parametro di ensure()

```
def ensure(name, validate, doc=None):
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
        return Class
    return decorator
```

decorator()

- riceve una classe come unico argomento e crea un nome privato e lo assegna privateName;
- crea una funzione getter che restituisce il valore associato alla property;
- crea una funzione setter che, nel caso in cui validate() non lanci un'eccezione, modifica il valore della property con il nuovo valore value, eventualmente creando l'attributo property se non esiste

Class Decorator

- Una volta che sono stati creati getter e setter, essi vengono usati per creare una nuova proprietà che viene aggiunta come attributo alla classe passata come argomento a decorator().
- La proprietà viene creata invocando property() nell'istruzione evidenziata:
  - in questa istruzione viene invocata la funzione built-in setattr() per associare la proprietà alla classe
  - La proprietà così creata avrà nella classe il nome pubblico corrispondente al parametro name di ensure()

```
def ensure(name, validate, doc=None):
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
        return Class
    return decorator
```

Class Decorator

- Qualche considerazione sulle funzioni di validazione:
- la funzione di validazione is_in_range() usata per price e per quantity è una factory function che restituisce una nuova funzione is_in_range() che ha i valori minimo e massimo codificati al suo interno

```
def is_in_range(minimum=None, maximum=None):
    assert minimum is not None or maximum is not None
    def is_in_range(name, value):
        if not isinstance(value, numbers.Number):
            raise ValueError("{} must be a number".format(name))
        if minimum is not None and value < minimum:
            raise ValueError("{} {} is too small".format(name, value))
        if maximum is not None and value > maximum:
            raise ValueError("{} {} is too big".format(name, value))
    return is_in_range
```

- AssertionError se nessuno tra minimum o maximum è diverso da None
- ValueError se value non è un numero, se minimum è diverso da None e value < minimum, oppure se maximum è diverso da None e value>maximum

Class Decorator

- Questa funzione di validazione è usata per la proprietà title e ci assicura che il titolo non sia una stringa vuota.
  - Il nome di una proprietà è utile nei messaggi di errore: nell'esempio viene sollevata l'eccezione ValueError se name non è una stringa o se è una stringa vuota e il nome della proprietà compare nel messaggio di errore.

```
def is_non_empty_str(name, value):
    if not isinstance(value, str):
        raise ValueError("{} must be of type str".format(name))
    if not bool(value):
        raise ValueError("{} may not be empty".format(name))
```

Class Decorator

```
@do_ensure
class Book:

    title = Ensure(is_non_empty_str)
    isbn = Ensure(is_valid_isbn)
    price = Ensure(is_in_range(1, 10000))
    quantity = Ensure(is_in_range(0, 1000000))

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

- Applicare molti decoratori in sequenza è una pratica che non è accettata da tutti i programmatori
- In questo esempio, le 4 proprietà vengono create come istanze della classe Ensure
- __init__ della classe Book associa le proprietà all'istanza di Book creata
- il decoratore di classe @do_ensure rimpiazza ciascuna delle 4 istanze di Ensure con una proprietà con lo stesso nome dell'istanza. La proprietà avrà come funzione di validazione quella passata ad Ensure()

Class Decorator

- La classe Ensure è usata per memorizzare
  - la funzione di validazione che sarà usata dal setter della proprietà
  - l'eventuale docstring della proprietà
- Ad esempio, l'attributo title di Book è inizialmente creato come un'istanza di Ensure ma dopo la creazione della classe Book il decoratore @do_ensure rimpiazza ogni istanza di Ensure con una proprietà. Il setter usa la funzione di validazione con cui l'istanza è sta creata.

```
class Ensure:

    def __init__(self, validate, doc=None):
        self.validate = validate
        self.doc = doc
```

Class Decorator

- Il decoratore di classe do_ensure consiste di tre parti:
  - La prima parte definisce la funzione innestata make_property(). La funzione make_property() prende come parametro name (ad esempio, title) e un attributo di tipo Ensure e crea una proprietà il cui valore viene memorizzato in un attributo privato (ad esempio, "_title"). Il setter al suo interno invoca la funzione di validazione.

```
def do_ensure(Class):
    def make_property(name, attribute):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            attribute.validate(name, value)
            setattr(self, privateName, value)
        return property(getter, setter, doc=attribute.doc)
    for name, attribute in Class.__dict__.items():
        if isinstance(attribute, Ensure):
            setattr(Class, name, make_property(name, attribute))
    return Class
```

Class Decorator

- La seconda parte itera sugli attributi della classe e rimpiazza ciascun attributo di tipo Ensure con una nuova proprietà con lo stesso nome dell'attributo rimpiazzato.
- La terza parte restituisce la classe modificata

```
def do_ensure(Class):
    def make_property(name, attribute):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            attribute.validate(name, value)
            setattr(self, privateName, value)
        return property(getter, setter, doc=attribute.doc)
    for name, attribute in Class.__dict__.items():
        if isinstance(attribute, Ensure):
            setattr(Class, name, make_property(name, attribute))
    return Class
```

Class Decorator

- In teoria avremmo potuto evitare le funzioni innestate e porre il codice di quelle funzioni dopo il test isinstance().
- Ciò non avrebbe però funzionato in pratica a causa di problemi con il binding ritardato.
- Questo problema si presenta abbastanza frequentemente quando si creano decoratori o decorator factory.
  - In genere per risolvere il problema è sufficiente usare una funzione separata (eventualmente innestata)

Class Decorator nella derivazioni di classi

- A volte creiamo una classe di base con metodi o dati al solo scopo di poterla derivare più volte.
- Ciò evita di dover duplicare i metodi o i dati nelle sottoclassi ma se i metodi ereditati o i dati non vengono mai modificati nelle sottoclassi, è possibile usare un decoratore di classe per raggiungere lo stesso obiettivo.

Class Decorator nella derivazioni di classi

- Questa è la classe base che verrà estesa da classi che non modificano il metodo ereditato on_change() method.

```
class Mediated:
    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```


Class Decorator nella derivazioni di classi

Possiamo applicare il decoratore di classe definito in basso in questo modo:

@mediated
class Button: ...

La classe Button avrà esattamente lo stesso comportamento che avrebbe avuto se l'avessimo definita come sottoclasse di Mediated con

class Button(Mediated): ...

```
def mediated(Class):
    setattr(Class, "mediator", None)
    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
    setattr(Class, "on_change", on_change)
    return Class
```