Programmazione Avanzata

Design Pattern: Observer

1

IL Design Pattern Observer

- Il pattern Observer e` un design pattern comportamentale che supporta relazioni di dipendenza many-to-many tra oggetti in modo tale che quando un oggetto cambia stato tutti gli oggetti collegati sono informati del cambio.

2

IL Design Pattern Observer: un esempio

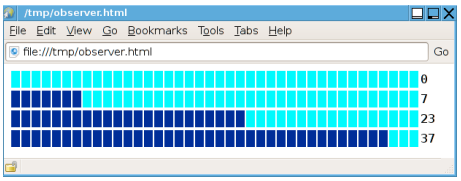
- Uno degli esempi di questo pattern e delle sue varianti e` il paradigma model/view/controller (MVC) che consiste nel separare un'applicazione in tre componenti logiche: modello, view e controller.
  - Il modello gestisce i dati e la logica dell'applicazione indipendentemente dall'interfaccia utente.
  - Una o piu` view visualizzano i dati in una o piu` forme comprensibili per l'utente. Ogni cambio nel modello si riflette automaticamente nelle view associate.
  - Uno o piu` controller mediano tra input e modello, cioe` convertono l'input in comandi per il modello o le view..
- Una popolare semplificazione dell'approccio MVC consiste nell'usare un paradigma model/view dove le view si occupano sia di visualizzare i dati sia di mediare tra input e modello.
  - In termini di Observer Pattern cio` significa che le view sono osservatori del modello e il modello e` l'oggetto dell'osservazione.

3

IL Design Pattern Observer: un esempio

- Consideriamo un modello che rappresenta un valore con un minimo e un massimo, come ad esempio una scrollbar o un controllo della temperatura.
- Vengono creati due osservatori (view) separati per il modello: uno per dare in output il valore del modello ogni volta che esso cambia sotto forma di una barra di progressione in formato HTML, l'altro per mantenere la storia dei cambiamenti (valori e timestamp).

```
$ ./observer.py > /tmp/observer.html
  0 2013-04-09 14:12:01.043437
  7 2013-04-09 14:12:01.043527
 23 2013-04-09 14:12:01.043587
 37 2013-04-09 14:12:01.043647
```



4

IL Design Pattern Observer: un esempio

- La classe Observed e` estesa dai modelli o da ogni altra classe che supporta l'osservazione.
- La classe Observed mantiene un insieme di oggetti osservatori.
  - Ogni volta che viene aggiunto un oggetto osservatore all'oggetto osservato, il metodo update() dell'osservatore e` invocato per inizializzare l'osservatore con lo stato attuale del modello.
  - Se in seguito il modello cambia stato, esso invoca il metodo ereditato observers_notify() in modo tale che il metodo update() di ogni osservatore possa essere invocato per assicurare che ogni osservatore (view) rappresenti il nuovo stato del modello.

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.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)
```

5

IL Design Pattern Observer: un esempio

- Il metodo observers_add()
  - accetta uno o piu` osservatori da aggiungere. Per questo motivo oltre a *observer c'e` il parametro observer che assicura che il numero di osservatori passati in input al metodo non sia zero.
  - usa nel for il metodo itertools.chain(*iterables) che crea un iteratore che restituituisce gli elementi dall'oggetto iterabile specificato come primo argomento e quando non ci sono piu` elementi da restituire in questo oggetto iterabile, passa al prossimo oggetto iterabile specificato come argomento e cosi` via fino a che non vengono restituiti gli elementi di tutte le collezioni iterabili in iterables. Il for avrebbe potuto usare la concatenazione di tuple in questo modo "for observer in (observer,) + observers:"

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.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)
```

6

IL Design Pattern Observer: un esempio

- La classe SliderModel eredita dalla classe Observed un insieme privato di osservatori che inizialmente e` vuoto e i metodi observers_add(), observer_discard() e observers_notify()
- Quando lo stato del modello cambia, per esempio quando il suo valore cambia, esso deve invocare il metodo observers_notify() in modo che ciascun osservatore possa rispondere di conseguenza.
- SliderModel ha anche le proprieta` minimum e maximum i cui setter, come quello di value, invocano il metodo observers_notify().

```
class SliderModel(Observed):

    def __init__(self, minimum, value, maximum):
        super().__init__()
        # These must exist before using their property setters
        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()
    ...
```


7

IL Design Pattern Observer: un esempio

- HistoryView e` un osservatore del modello e per questo fornisce un metodo update() che accetta il modello osservato come suo unico argomento (oltre self).
- Ogniqualvolta il metodo update() e` invocato, esso aggiunge una tupla (value, timestamp) alla sua self.data list, mantenendo in questo modo la storia di tutti i cambiamenti applicati al modello.

```
class HistoryView:
    def __init__(self):
        self.data = []

    def update(self, model):
        self.data.append((model.value, time.time()))
```


8

IL Design Pattern Observer: un esempio

- Questa e` un'altra view per osservare il modello. L'attributo length rappresenta il numero di celle usate per rappresentare il valore del modello in una riga della tabella HTML.
- Il metodo update viene invocato quando il modello e` osservato per la prima volta e quando viene successivamente aggiornato
  - Il metodo stampa una tabella HTML di una riga con un numero self.length di celle per rappresentare il modello. Le celle sono di colore ciano se sono vuote e blu scuro altrimenti.

```
class LiveView:
    def __init__(self, length=40):
        self.length = length
```

```
    def update(self, model):
        tippingPoint = round(model.value * self.length /
                (model.maximum - model.minimum))
        td = '<td style="background-color: {}"> </td>'
        html = ['<table style="font-family: monospace" border="0"><tr>']
        html.extend(td.format("darkblue") * tippingPoint)
        html.extend(td.format("cyan") * (self.length - tippingPoint))
        html.append("<td>{}</td></tr></table>".format(model.value))
        print("".join(html))
```


9

IL Design Pattern Observer: un esempio

- Il main() comincia con il creare due view, HistoryView e LiveView.
- Poi crea un modello con minimo 0, valore attuale 0 e massimo 40 e rende le due view osservatori del modello
- Non appena viene aggiunta LiveView come observer del modello, essa produce il suo primo output e non appena viene aggiutno HistoryView, esso registra il suo primo valore e il suo primo timestamp.
- Poi viene aggiornato il valore del modello tre volte e ad ogni aggiornamento LiveView restituisce una nuova tabella di una riga e HistoryView registra il valore e il timestamp .

```
def main():
    historyView = HistoryView()
    liveView = LiveView()
    model = SliderModel(0, 0, 40) # minimum, value, maximum
    model.observers_add(historyView, liveView)  # liveView produces output
    for value in (7, 23, 37):
        model.value = value                     # liveView produces output
    for value, timestamp in historyView.data:
        print("{:3} {}".format(value, datetime.datetime.fromtimestamp(
                timestamp)), file=sys.stderr)
```


10

Programmazione Avanzata

Design Pattern: Prototype

Il pattern Prototype

- Il pattern Prototype è un design pattern creazionale usato per creare nuovi oggetti clonando un oggetto preesistente e poi modificando il clone così creato.

Il pattern Prototype: esempio

- Point è la classe preesistente

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```


13

Il pattern Prototype: esempio

- In questo codice cloniamo nuovi punti in diversi modi

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14) # Could have used any of point1 to point6
```


14

Il pattern Prototype: esempio

- point1 = Point(1, 2)
  - viene semplicemente invocato il costruttore della classe Point.
  - point 1 è creato in modo statico. Di seguito creeremo istanze di Point in modo dinamico.
- point2 = **eval**("{}({}, {})".format("Point", 2, 4))
  - usa eval() per creare istanze di Point
  - eval valuta l'espressione Python rappresentata dalla stringa ricevuta come argomento. In altre parole, esegue il codice passato come argomento

15

Il pattern Prototype: esempio

- point3 = **getattr**(sys.modules[__name__], "Point")(3, 6)
  - usa getattr() per creare un'istanza
  - **getattr(object,name, default)** restituisce il valore dell'attributo dell'oggetto
    - **object** : oggetto per il quale viene restituito il valore dell'attributo nominato
    - **name** : stringa che contiene il nome dell'attributo
    - **default (opzionale):** valore restituito quando l'attributo specificato non viene trovato
  - nel codice in alto
  - **sys.modules** è un dizionario che mappa i nomi dei moduli ai moduli che sono già stati caricati
  - l'espressione sys.modules[__name__] restituisce il modulo in cui essa si trova
  - l'espressione **getattr**(sys.modules[__name__], "Point") restituisce il valore dell'attributo Point del modulo

16

Il pattern Prototype: esempio

- point4 = **globals**()["Point"](4, 8)

  - La funzione built-in globals() restituisce un dizionario che rappresenta la tavola dei simboli globali. All'interno di una funzione o un metodo, il dizionario è quello relativo al modulo dove è definita la funzione (o il metodo), non quello in cui è invocata .
  - comportamento similie a **getattr(object,name, default)**

17

Il pattern Prototype: esempio

- point5 = **make_object**(Point, 3,9)

  - usa la funzione make-object

18

Il pattern Prototype: esempio

- point6 = copy.deepcopy(point5)
  - usa il classico approccio basato su Prototype:
    - prima clona un oggetto esitente
    - poi lo inizializza con le istruzioni successive

- point7 = point1.__class__(7, 14)
  - point7 è creato usando point1
  - instanza.__class__ contiene la classe a cui appartiene istanza

Python ha un support built-in per creare oggetti sulla base di prototipi: la funzione copy.deepcopy().

Il pattern Prototype

- Il pattern Prototype è usato per creare nuovi oggetti clonando un oggetto preesistente e poi modificando il clone così creato.

Il pattern Prototype: esempio

- Point è la classe preesistente

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```


21

Il pattern Prototype: esempio

- In questo codice cloniamo nuovi punti in diversi modi

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14) # Could have used any of point1 to point6
```


22

Il pattern Prototype: esempio

- point3 = **getattr**(sys.modules[__name__], "Point")(3, 6)
  - usa getattr() per creare un'istanza
  - **getattr(object,name, default)** restituisce il valore dell'attributo dell'oggetto
    - **object** : oggetto per il quale viene restituito il valore dell'attributo nominato
    - **name** : stringa che contiene il nome dell'attributo
    - **default (opzionale):** valore restituito quando l'attributo specificato non viene trovato
  - nel codice in alto
  - **sys.modules** è un dizionario che mappa i nomi dei moduli ai moduli che sono già stati caricati
  - l'espressione sys.modules[__name__] restituisce il modulo in cui essa si trova
  - l'espressione **getattr**(sys.modules[__name__], "Point") resituisce il valore dell'attributo Point del modulo

23

Il pattern Prototype: esempio

- point4 = **globals**()["Point"](4, 8)
  - La funzione built-in globals() restituisce un dizionario che rappresenta la tavola dei simboli globali. All'interno di una funzione o un metodo, il dizionario è quello relativo al modulo dove è definita la funzione (o il metodo), non quello in cui è invocata .
  - comportamento similie a **getattr(object,name, default)**

24

Il pattern Prototype: esempio

- point1 = Point(1, 2)
  - viene semplicemente invocato il costruttore della classe Point
- point2 = **eval**("{}({}, {})".format("Point", 2, 4))
  - usa eval() per creare istanze di Point
  - eval valuta l'espressione Python rappresentata dalla stringa ricevuta come argomento. In altre parole, esegue il codice passato come argomento

eval

- https://docs.python.org/3/library/functions.html#eval
- Gli argomenti sono un stringa e due argomenti opzionali *globals* and l*ocals*.
- *globals* deve essere un dizionario mentre *locals* puo` essere di un qualsiasi tipo mapping
- La stringa passato come argomento viene valutata come un'espressione Python usando i dizionari *globals* and *locals* dictionaries come namespace globali e locali.
- Se in *globals* non è presente un valore per la chiave __builtins__, un riferimento al modulo built-in builtins è associato alla chiave __builtins__ prima di fare il parsing dell'espressione.
- Il valore di default di *locals* è il dizionario *globals.*
- Se entrambi i dizionari *globals* and *locals* sono omessi l'espressione è eseguita con i *globals* and *locals* nell'ambiente in cui eval() è invocata*.*

eval

```
>>> x = 4
>>> y = 3
# valuta l'espressione x + y
>>> s = eval('x + y')
>>> print('s: ', s)
s: 7
>>> # la prossima invocazione di eval usa il parametro globals al posto del namespace globale
>>> t = eval('x + y', {'x': 5, 'y': 8})
>>> print('t: ', t)
t: 13
>>> # la prossima invocazione di eval usa il valore globale di x uguale a 5 e quello locale di y uguale a 9
>>> r = eval('x + y', {'x': 5, 'y': 8}, {'y': 9, 'w': 2})
>>> print('r: ', r)
r: 14
>>> p = eval('print(x, y)')
4 3
>>> print('p: ', p)
p: None
```


Il pattern Prototype: esempio

- point5 = **make-object**(Point, 3,9)
  - usa la funzione make-object

Il pattern Prototype: esempio

- point6 = copy.deepcopy(point5)
  - usa il classico approccio basato su Prototype:
    - prima clona un oggetto esitente
    - poi lo inizializza con le istruzioni successive

- point7 = point1.__class__(7, 14)
  - point7 è creato usando point1
  - instanza.__class__ contiene la classe a cui appartiene istanza

- Python ha un support built-in per creare oggetti sulla base di prototipi: la funzione copy.deepcopy().

29

Programmazione Avanzata

Design Pattern: Flyweight

30

Il Design Pattern Flyweight

- Il pattern Flyweight è concepito per gestire un grande numero di oggetti relativamente piccoli dove molti degli oggetti sono duplicati l'uno dell'altro.
- Il pattern è implementato in modo da avere un'unica istanza per rappresentare tutti gli oggetti uguali tra loro. Ogni volta che è necessario, questa unica istanza viene condivisa.
- Python permette di implementare Flyweight in modo naturale grazie all'uso dei riferimenti. Ad esempio, una lunga lista di stringhe molte delle quali sono duplicati potrebbe richiedere molto meno spazio se al posto delle stringhe venissero memorizzati i riferimenti ad esse.

```
red, green, blue = "red", "green", "blue"
x = (red, green, blue, red, green, blue, red, green)
y = ("red", "green", "blue", "red", "green", "blue", "red", "green")
```

Nel codice in alto, x immagazzina 3 stringhe usando 8 riferimenti mentre la tupla y immagazzina 8 stringhe usando 8 riferimenti dal momento che quello che abbiamo scritto corrisponde a _anonymous_item0 = "red", …, _anonymous_item7 = "green"; y = (_anonymous_item0, …_anonymous_item7).

31

Il Design Pattern Flyweight

- Probabilmente il modo più semplice per trarre vantaggio dal pattern Flyweight in Python è di usare un dict, in cui ciascun oggetto (unico) corrisponde ad un valore identificato da un'unica chiave.
- Ciò assicura che ciascun oggetto distinto viene creato un'unica volta, indipendentemente da quante volte viene usato.
- In alcune situazioni si potrebbero avere molti oggetti non necessariamente piccoli dove gran parte di essi o tutti sono unici. Un facile modo per ridurre l'uso della memoria in questo è di usare __slots__

32

__slots__

- In Python ogni classe può avere attributi di istanza.
- Per default Python usa un dict per immagazzinare gli attributi di instanza di un oggetto. Ciò è molto utile perché consente di settare nuovi attributi durante l'esecuzione.
- Comunque per classi piccole con attributi noti questo comportamento potrebbe essere un collo di bottiglia in quanto il dict comporterebbe uno spreco di RAM nel caso in cui vengano creati molti oggetti.

__slots__

- Un modo per evitare questo spreco di RAM e di usare __slots__ per indicare a Python di non usate un dict, e di allocare spazio solo per un insieme fissato di attributi.

- __slots__ è una variabile di classe a cui può essere assegnata una stringa, un iterabile, o una sequenza di stringhe.
- *__slots__* riserva spazio per le variabili dichiarate e previene la creazione automatica di *__dict__* (e di *__weakref__)* per ciascuna istanza.

__slots__

- Vediamo un esempio di implementazione della stessa classe con e senza __slots__ .
- Senza __slots__

```
class MyClass():
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
```


35

__slots__

- con __slots__

```
class MyClass():
    __slots__ = [‘name’, 'identifier’]
    def __init__(self, name, identifier):
        self.name = name
        self.identifier=identifier
```

Non posso aggiungere altre variabili di istanza alle istanze di MyClass

36

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ("x", "y", "z", "color")

    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

- La classe Point mantiene una posizione nello spazio tridimensionale e un colore.
- Grazie a __slots__, nessun Point ha il suo dict (self.__dict__) privato.
- Ciò vuol dire che nessun attributo può essere aggiunto a punti individuali.
- Un programma per creare una tupla di un milione di punti ha impiegato su una stessa macchina
- nella versione con slots, circa 2 secondi e il programma ha occupato 183 Mebibyte di RAM
- nella versione senza slots, una frazione di secondo in meno ma il programma ha occupato 312 Mebibyte di RAM.

- Per default Python sacrifica sempre la memoria a favore della velocità ma è sempre possibile invertire queste priorità se è conveniente farlo.

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

Questo è l'inizio di un'altra classe Point.
Essa utilizza un database DBM (chiave-valore) immagazzinato in un file su disco.
Un riferimento al DBM è mantenuto nella variabile Point.__dbm.
Tutti i punti condividono lo stesso file DBM.

Uno "shelf" è un oggetto persistente simile ad un dizionario. I valori (non le chiavi) in uno shelf possono essere arbitrari oggetti gestibili dal modulo pickle. Ciò include la maggior parte di istanze di classi, tipi di dati ricorsivi, e oggetti contenenti molti oggetti condivisi.
Le chiavi sono stringhe.
shelve.open(*filename*, *flag='c'*, *protocol=None*, *writeback=False*) apre un dizionario persistente.
Il filename specificato è il nome di base per il database sottostante.
Per default il file database è aperto in lettura e scrittura.

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

tempfile.gettempdir() restituisce il nome della directory usata per i file temporanei.

Il comportamento di default di open fa in modo che venga creato il file DBM se non esiste già .
Il modulo shelve serializza i valori immagazzinati e li deserializza quando i valori vengono recuperati dal database.
Il processo di deserializzazione in Python non è sicuro perché esegue dell'arbitrario codice Python e di conseguenza non dovrebbe mai essere effettuato su dati provenienti da fonti non affidabili,

Il Design Pattern Flyweight

```
    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

A differenza del metodo __init__() della prima classe Point, questo metodo assegna i valori delle variabili in un file DBM.

```
    def __getattr__(self, name):
        return Point.__dbm[self.__key(name)]
```

Questo metodo è invocato ogni volta che si accede ad un attributo della classe

Il Design Pattern Flyweight

```
def __key(self, name):
    return  "{:X}:{}".format(id(self), name)
```

Questo metodo fornisce una stringa chiave per ognuno degli attributi x, y, z e color.
La chiave è ottenuta dall'ID restituita da id(self) in esadecimale e dal nome dell'attributo. Per esempio se l'ID di un punto è 3954827, il suo attributo x avrà chiave "3C588B:x", il suo attributo y avrà chiave "3C588B:y", e così via.

Il design pattern Flyweight

- Le chiavi e i valori dei database DBM devono essere byte.
- Per fortuna, i moduli DBM Python accettano sia str che byte come chiavi convertendo le stringhe in byte.
- In particolare, il modulo shelve, qui usato, permette di immagazzinare un qualsiasi valore gestibile dal modulo pickle.
- Un valore recuperato dal database è convertito dalla rappresentazione sotto forma di sequenza di bytes nel tipo originario.

Il Design Pattern Flyweight

```
def __setattr__(self, name, value):
    Point.__dbm[self.__key(name)] = value
```

Ogni volta che un attributo di Point è settato (ad esempio, point.y = y), viene invocato questo metodo.
Il valore value immagazzinato è convertito in un flusso di byte.

Il Design Pattern Flyweight

Sulla macchina usata per i test, la creazione di un milione di punti ha richiesto circa un minuto ma il programma ha occupato solo 29 Mebibyte of RAM (più 361 Mebibyte di spazio su disco) mentre la prima versione di Point ha richiesto 183 Mebibyte di RAM.