Programmazione Avanzata

Design Pattern: State

Il Design Pattern State

State è un design pattern comportamentale che consente ad un oggetto di modificare il proprio comportamento quando il suo stato interno cambia

Utile nei seguenti casi:

- Il comportamento di un oggetto dipende dal suo stato e deve cambiare comportamento durante l'esecuzione del programma in base al suo stato
- Le operazioni contengono statement condizionali grandi che dipendono dallo stato dell'oggetto. Lo stato dell'oggetto è di solito rappresentato da una o più costanti numerate. Il pattern State inserisce ciascun caso dello statement condizionale in una classe separata.
  - **Ciò consente di trattare lo stato dell'oggetto come un vero e proprio oggetto che può cambiare indipendentemente da altri oggetti.**

Il Design Pattern State: un semplice esempio

```
class State_d:
    def __init__(self, imp):
        self.__implementation = imp
    def changeImp(self, newImp):
        self.__implementation = newImp
    # Delegate calls to the implementation:
    def __getattr__(self, name):
        return getattr(self.__implementation, name)

class Implementation1:
    def f(self):
        print("Fiddle de dum, Fiddle de dee,")
    def g(self):
        print("Eric the half a bee.")
    def h(self):
        print("Ho ho ho, tee hee hee,")

class Implementation2:
    def f(self):
        print("We're Knights of the Round Table.")
    def g(self):
        print("We dance whene'er we're able.")
    def h(self):
        print("We do routines and chorus scenes")
```

State è simile a Proxy ma a differenza di Proxy utilizza più implementazioni ed un metodo per passare da un'implementazione all'altra durante la vita del surrogato.

#Uso di State_d

```
def run(b):
    b.f()
    b.g()
    b.h()
    b.g()

b = State_d(Implementation1())
run(b)
b.changeImp(Implementation2())
run(b)
```


16

Il Design Pattern State: un esempio

Consideriamo una classe multiplexer che ha due stati che influiscono sul comportamento dei metodi della classe.

attivo: il multiplexer accetta connessioni, cioè coppie (nome evento, callback), dove callback è un qualsiasi callable. Dopo che sono state stabilite le connessioni, ogni volta che viene inviato un evento al multiplexer, i callback associati all'evento vengono invocati.

dormiente: l'invocazione dei suoi metodi non ha alcun effetto (comportamento safe)

Nell'esempio vengono create delle funzioni callback che contano il numero di eventi che ricevono. Queste funzioni vengono connesse ad un multiplexer attivo. Poi vengono inviati un certo numero di eventi random al multiplexer e stampati i conteggi tenuti dalle funzioni callback.

17

Il Design Pattern State: un esempio

- Dopo aver inviato 100 eventi random al multiplexer attivo, lo stato del multiplexer viene cambiato in dormiente e gli vengono inviati altri 100 eventi random, ciascuno dei quali deve essere ignorato.
- Il multiplexer è quindi riportato nello stato attivo e gli vengono inviati altri eventi ai quali il multiplexer deve rispondere invocando i callback associati.
- si veda anche il codice completo in multiplexer1.py

```
$ ./multiplexer1.py
After 100 active events:  cars=150 vans=42 trucks=14 total=206
After 100 dormant events: cars=150 vans=42 trucks=14 total=206
After 100 active events:  cars=303 vans=83 trucks=30 total=416
```

output del programma

Il Design Pattern State: un esempio

- Il main() comincia con il creare dei contatori. **Le istanze così create sono callable** e quindi possono essere usate come funzioni.
  - Le istanze di Counter mantengono contatori separati per ciascuno dei nomi passati come argomento o, in assenza di un nome (come totalCounter), matengono un contatore singolo.
- Viene quindi creato un multiplexer (che per default è attivo) e vengono connesse le funzioni callback agli eventi.
- I nomi degli eventi considerati sono “cars”, “vans” e “trucks”.
  - Nel for, la funzione carCounter() è connessa all’evento “cars”, la funzione commercialCounter() è connessa agli eventi “vans” e “trucks” e totalCounter() è connessa a tutti e tre gli eventi.

```
totalCounter = Counter()
carCounter = Counter("cars")
commercialCounter = Counter("vans", "trucks")

multiplexer = Multiplexer()
for eventName, callback in (("cars", carCounter),
        ("vans", commercialCounter), ("trucks", commercialCounter)):
    multiplexer.connect(eventName, callback)
    multiplexer.connect(eventName, totalCounter)
```

Il Design Pattern State: un esempio

- Per un evento “cars”, il multiplexer invoca carCounter() e totalCounter(), passando l’evento come unico argomento a ciascuna chiamata. Se l’evento è invece “vans” o “trucks”, il multiplexer invoca le funzioni commercialCounter() e totalCounter().

- Con il codice mostrato in basso, main() genera 100 eventi random e li invia al multiplexer.

```
for event in generate_random_events(100):
    multiplexer.send(event)
print("After 100 active events:  cars={} vans={} trucks={} total={}"
        .format(carCounter.cars, commercialCounter.vans,
                commercialCounter.trucks, totalCounter.count))
```


20

Il Design Pattern State: un esempio

Il metodo __init__() della classe Counter:

- Se non vengono forniti nomi, viene creata un’istanza di un contatore anonimo il cui conteggio è mantenuto in self.count;
- In caso contrario, vengono mantenuti conteggi indipendenti mediante la funzione built-in setattr() per ciascuno dei nomi passati a __init__().
- Ad esempio, per l’istanza carCounter viene creato l’attributo self.cars.

```
class Counter:

    def __init__(self, *names):
        self.anonymous = not bool(names)
        if self.anonymous:
            self.count = 0
        else:
            for name in names:
                if not name.isidentifier():
                    raise ValueError("names must be valid identifiers")
                setattr(self, name, 0)
```


21

Il Design Pattern State: un esempio

- Il metodo __call__() della classe Counter
- Quando un'istanza di Counter è invocata, la chiamata è passata a __call__()
- Se il contatore è anonimo, self.count viene incrementato; altrimenti si cerca di recuperare l'attributo corrispondente al nome dell'evento.
  - Ad esempio, se il nome dell'evento è "trucks", count viene settato con il valore di self.trucks. Viene quindi aggiornato il valore dell'attributo con il vecchio conteggio più il nuovo conteggio dell'evento.
- Siccome non è fornito un valore di default per la funzione built-in gettattr(), se l'attributo non esiste viene lanciato un AttributeError. Ciò assicura anche che non venga creato un attributo con un nome sbagliato perché in caso di errore la chiamata a setattr() non viene raggiunta.

```
    def __call__(self, event):
        if self.anonymous:
            self.count += event.count
        else:
            count = getattr(self, event.name)
            setattr(self, event.name, count + event.count)
```


22

Il Design Pattern State: un esempio

La classe Event è molto semplice perché serve esclusivamente come parte dell'infrastruttura per illustrare il Pattern State.

```
class Event:

    def __init__(self, name, count=1):
        if not name.isidentifier():
            raise ValueError("names must be valid identifiers")
        self.name = name
        self.count = count
```


23

Il Design Pattern State: un esempio

La classe Multiplexer:

- Ci sono due approcci principali che possiamo utilizzare per gestire lo stato interno ad una classe.
  - Approccio che usa **metodi state-sensitive** (comportamento dei metodi si adatta allo stato)
  - Approccio che usa **metodi state-specific** (progettati ad hoc per specifici stati)

- Consideriamo il primo dei due approcci:

```
class Multiplexer:

    ACTIVE, DORMANT = ("ACTIVE", "DORMANT")

    def __init__(self):
        self.callbacksForEvent = collections.defaultdict(list)
        self.state = Multiplexer.ACTIVE
```


24

Il Design Pattern State: un esempio

- La classe collections.defaultdict (https://docs.python.org/3.7/library/collections.html#collections.defaultdict):
- La classe defaultdict è una sottoclasse di dict.
- Il costruttore riceve come primo argomento un valore per il suo attributo default_factory (per default è None), usato per creare valori non presenti nel dizionario.
- I restanti argomenti corrispondono a quelli passati al costruttore di dict.
- Se default_factory non è None, esso viene invocato senza argomenti per fornire un valore di default per una data chiave k. Tale valore viene inserito nel dizionario (associato alla chiave k) e restituito.
- Questo metodo è invocato dal metodo __getitem__() quando la chiave richiesta non viene trovata.
- Gli altri metodi non invocano default_factory per cui get() restituisce None se la chiave non è nel dizionario.

25

Il Design Pattern State: un esempio

- La classe Multiplexer ha due possibili stati: ACTIVE e DORMANT.
  - ACTIVE: i metodi state-sensitive svolgono un lavoro utile
  - DORMANT: i metodi state-sensitive non fanno niente.
- Un nuovo Multiplexer è creato nello stato ACTIVE.
- self.callbacksForEvent è un dizionario (di tipo defaultdict) di coppie (nome evento, lista di callable)
- **Il metodo connect è usato per creare un'associazione tra un evento con un certo nome e un callback.**
- Se il nome dell'evento non è nel dizionario, il fatto che self.callbacksForEvent sia un defaultdict garantisce che venga creato un elemento con chiave uguale al nome dell'evento e con valore uguale ad una lista vuota che verrà poi restituita.
- Se il nome dell'evento è già nel dizionario, verrà restituita la lista associata.
- In entrambi i casi , con append() viene poi aggiunta alla lista il callback da associare all'evento

```
def connect(self, eventName, callback):
    if self.state == Multiplexer.ACTIVE:
        self.callbacksForEvent[eventName].append(callback)
```

26

Il Design Pattern State: un esempio

Se invocato senza specificare un callback, questo metodo disconnette tutti i callback associati con il nome dell'evento dato; altrimenti rimuove solo il callback specificato dalla lista dei callback associata al nome dell'evento.

```
def disconnect(self, eventName, callback=None):
    if self.state == Multiplexer.ACTIVE:
        if callback is None:
            del self.callbacksForEvent[eventName]
        else:
            self.callbacksForEvent[eventName].remove(callback)
```


27

Il Design Pattern State: un esempio

Se un evento è inviato al multiplexer e questo è attivo allora send itera su tutti i callback associati all'evento (se ve ne sono) e invoca ciascuno di questi callback passandogli l'evento come argomento.

```
def send(self, event):
    if self.state == Multiplexer.ACTIVE:
        for callback in self.callbacksForEvent.get(event.name, ()):
            callback(event)
```


Il Design Pattern State: un esempio

Approccio basato su metodi state-specific:

- La classe Multiplexer ha gli stessi due stati di prima e lo stesso metodo __init__. Questa volta però l'attributo self.state è una proprietà.
- Questa versione di multiplexer non immagazzina lo stato come tale ma lo computa controllando se uno dei metodi pubblici è stato settato ad un metodo privato attivo o passivo.

```
@property
def state(self):
    return (Multiplexer.ACTIVE if self.send == self.__active_send
            else Multiplexer.DORMANT)
```

Il Design Pattern State: un esempio

- Ogni volta che viene cambiato lo stato, il setter della proprietà associa il multiplexer ad un insieme di metodi appropriati al suo stato
- Se, ad esempio, lo stato è DORMANT, ai metodi pubblici viene assegnata la versione lambda dei metodi

```
@state.setter
def state(self, state):
    if state == Multiplexer.ACTIVE:
        self.connect = self.__active_connect
        self.disconnect = self.__active_disconnect
        self.send = self.__active_send
    else:
        self.connect = lambda *args: None
        self.disconnect = lambda *args: None
        self.send = lambda *args: None
```


30

Il Design Pattern State: un esempio

- _active_connect è un metodo privato che può essere assegnato al corrispondente metodo pubblico self.connect se lo stato del multiplexer è ACTIVE. I metodi _active_disconnect e _active_ send sono simili.
  - **Nessuno di questi tre metodi controlla lo stato dell'istanza.**

```
def __active_connect(self, eventName, callback):
    self.callbacksForEvent[eventName].append(callback)
```


31

Il Design Pattern State: esercizio

Immaginiamo che un bambino venga iscritto alla scuola media. Il bambino puo` essere in uno dei seguenti stati:

a. iscritto: il bimbo e` inizialmente iscritto al primo anno
b. alSecondoAnno: il bimbo e` promosso al secondo anno
c. alTerzoAnno: il bimbo e` promosso al terzo anno
d. diplomato: al termine del terzo, il bimbo consegue il diploma di scuola media.

La classe Bambino ha il metodo succ per passare allo stato successivo, il metodo pred per passare a quello precedente (retrocesso in caso di debiti formativi non recuperati) e il metodo salta_anno per saltare un anno (da iscritto si salta al terzo anno e dal secondo anno al diploma). Lo stato iscritto non ha stati che lo precedono; lo stato diplomato non ha stati che vengono dopo di esso.

La classe Bambino ha anche un metodo stampaStato per stampare lo stato del bambino. Scrivere la classe **Bambino** usando un approccio state-specific in cui lo stato del bambino e` una proprieta`. **Non usare altre classi oltre la classe Bambino.**