Programmazione Avanzata

Design Pattern: Chain of Responsability

1

Il Pattern Chain of Responsibility

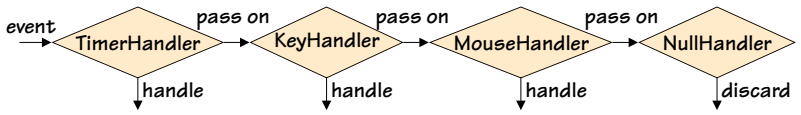
- Il Pattern Chain of Responsibility è un design pattern comportamentale ed è utilizzato per separare il codice che effettua una richiesta da quello che elabora la richiesta.
- Invece di avere una funzione che invoca direttamente un'altra funzione, la prima funzione invia la richiesta ad una catena di destinatari.
  - Il primo destinatario può o elaborare la richiesta o passare la richiesta al prossimo destinatario nella catena; il secondo destinatario si comporta allo stesso modo del primo e così via fino a che non viene raggiunto l'ultimo destinatario che può decidere se scartare la richiesta o lanciare un'eccezione.

2

Il Pattern Chain of Responsibility: esempio

- Immaginiamo di avere un'interfaccia utente che riceve un evento da gestire. Alcuni eventi provengono dall'utente, altri dal sistema (ad esempio eventi temporizzati).
- A ciascun evento corrisponde una classe per la sua gestione
- La seguente linea di codice mostra come creare una catena di 4 gestori (ciascun gestore è un;istanza di una diversa classe) per gestire eventi.
  - Siccome gli eventi non gestiti vengono scartati, l'argomento di MouseHandler avrebbe potuto essere None (o non esserci).
  - L'ordine in cui inseriamo i gestori nella catena non dovrebbe essere rilevante perche' ogni gestore deve gestire solo l'evento per cui è stato disegnato.

```
handler1 = TimerHandler(KeyHandler(MouseHandler(NullHandler())))
```



3

Il Pattern Chain of Responsibility: esempio

- Gli eventi sono normalmente gestiti in un loop.
- Nel codice seguente si esce dal loop e si termina l'applicazione se c'è un evento TERMINATE; altrimenti si passa l'evento alla catena che gestisce gli eventi.

```
while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    handler1.handle(event)
```

- Nel seguente codice si crea un nuovo gestore di eventi.
- DebugHandler deve essere il primo gestore della catena in quanto è usato per spiare e riportare gli eventi passati alla catena, non per gestirli.
- In alternativa, si può invocare handler2.handle(event) nel loop in alto, in modo da avere, in aggiunta ai normali gestori di eventi, un output di debugging e vedere gli eventi ricevuti.

```
handler2 = DebugHandler(handler1)
```


4

Il Pattern Chain of Responsibility: esempio

- NullHandler serve come classe base per i nostri gestori di eventi e fornisce l'infrastruttura per gestire gli eventi.
- Se un'istanza è creata con un successore allora quando questa istanza riceve un evento, esso passa semplicemente l'evento al successore.
- Se invece l'istanza non ha un successore, l'evento viene scartato.
- Questo è l'approccio standard usato nella programmazione GUI (graphical user interface), sebbene si possa facilmente lanciare l'eccezione per eventi non gestiti.

```
class NullHandler:

    def __init__(self, successor=None):
        self.__successor = successor

    def handle(self, event):
        if self.__successor is not None:
            self.__successor.handle(event)
```


5

Il Pattern Chain of Responsibility: esempio

- Siccome nella classe seguente non viene reimplementato il metodo __init__(), viene usato il metodo __init__() della classe base e di conseguenza la variabile self.__successor__ viene creata correttamente.
- La classe MouseHandler gestisce solo gli eventi appropriati (cioè, di tipo Event.MOUSE) e passa ogni altro tipo di evento al suo successore nella catena, se ve ne è uno.
- Le classi KeyHandler e TimerHandler (non mostrate) hanno la stessa struttura di MouseHandler. Queste classi differiscono solo per il tipo di eventi che gestiscono (ad esempio, Event.KEYPRESS e Event.TIMER) e il tipo di gestione che svolgono (cioè, stampano messaggi differenti).

```
class MouseHandler(NullHandler):

    def handle(self, event):
        if event.kind == Event.MOUSE:
            print("Click:   {}".format(event))
        else:
            super().handle(event)
```


6

Il Pattern Chain of Responsibility:esempio

- La classe DebugHandler è diversa dagli altri gestori in quanto non gestisce mai eventi. Il gestore di tipo DebugHandler deve essere il primo nella catena.

- Il metodo __init__ della classe riceve in input un file per scrivere al suo interno il report e quando accade un evento, riporta l'evento e poi lo passa avanti nella catena.

```
class DebugHandler(NullHandler):

    def __init__(self, successor=None, file=sys.stdout):
        super().__init__(successor)
        self.__file = file

    def handle(self, event):
        self.__file.write("*DEBUG*: {}\n".format(event))
        super().handle(event)
```


7

Il Pattern Chain of Responsability: esempio basato su coroutine

In Python, ogni funzione o metodo che contiene un'espressione yield è un generatore.
Un generatore può essere trasformato in una coroutine mediante il decoratore @coroutine e mediante l'uso di un loop infinito.

```
def coroutine(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        generator = function(*args, **kwargs)
        next(generator)
        return generator
    return wrapper
```

- La funzione wrapper invoca function una sola volta e cattura il generatore prodotto nella variabile generator. Questo generatore non è altro che la funzione originaria con gli argomenti e le variabili locali catturate nel suo stato.
- La funzione wrapper invoca poi next(generator) per arrivare alla prima espressione yield del generatore e restituisce il generatore (insieme al suo stato). Questo generatore è una coroutine pronta per ricevere un valore alla sua prima (o unica) epressione yield.

8

Il Pattern Chain of Responsability: esempio basato su coroutine

- Un generatore è una funzione o un metodo che contiene una o più espressioni yield invece che dei return.
- Ogni volta che viene raggiunto un yield, viene restituito un valore e l'esecuzione della funzione o del metodo è sospesa con il suo stato intatto.
- Quando il generatore è usato nuovamente, l'esecuzione riprende dallo statement successivo all'espressione yield (maggiori dettagli sui generatori in un gruppo di slide a parte).
- Una coroutine usa l'espressione yield allo stesso modo di un generatore ma ha un comportamento particolare in quanto esegue un loop infinito e comincia sospesa alla sua prima (e unica, nelle coroutine del nostro esempio) espressione yield, in attesa che venga inviato un valore.
- Nel caso vi sia un'unica espessione yield, una coroutine si comporta nel modo seguente. Se e quando viene inviato un valore con una send, la coroutine lo riceve come valore dell'espressione yield in cui è sospesa in quel momento. La coroutine riprende l'esecuzione e può poi fare qualsiasi computazione desideri nel corpo del ciclo e quando ha finito essa cicla ancora e di nuovo sospende l'esecuzione in attesa di un valore da parte dell'espressione yield.
  - I valori sono spinti in una coroutine invocando il metodo send() della coroutine.

9

Il Pattern Chain of Responsability: esempio basato su coroutine

Siccome possiamo sia ricevere che inviare valori ad una coroutine, possiamo usare questi valori per creare delle pipeline, quali le catene per gestire gli eventi.

Non abbiamo più bisogno di successor perché ora possiamo usare la sintassi del generatore Python.

10

Il Pattern Chain of Responsability: esempio basato su coroutine

```
pipeline = key_handler(mouse_handler(timer_handler()))
```

```
while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    pipeline.send(event)
```

- Come nell'approccio di prima, una volta che la catena è pronta a gestire eventi, essi vengono gestiti in un loop.
- Poiché ogni funzione è una coroutine (una funzione generatrice) essa ha il metodo send
- Ogni volta che c'è un evento da gestire, esso è inviato con send alla pipeline.
- Nell'esempio in alto, l'evento sarà inviato inizialmente alla coroutine key_handler() .
- Come nell'approccio di prima , l'ordine dei gestori non è importante.

11

Il Pattern Chain of Responsability: esempio basato su coroutine

```
@coroutine
def key_handler(successor=None):
    while True:
        event = (yield)
        if event.kind == Event.KEYPRESS:
            print("Press:   {}".format(event))
        elif successor is not None:
            successor.send(event)
```

```
@coroutine
def debug_handler(successor, file=sys.stdout):
    while True:
        event = (yield)
        file.write("*DEBUG*: {}\n".format(event))
        successor.send(event)
```


12

Esercizio

- Scrivere una funzione che prende in input una sequenza di richieste (liste di due interi) e passa ciascuna richiesta ad una catena di gestori ciascuno dei quali è una coroutine.
- Se il primo intero della lista ènell'intervallo [0,4] allora la richiesta viene gestita dal gestore Handler_04 che stampa "Richiesta {} gestita da Handler_04".
- Se il primo intero della lista ènell'intervallo [5,9] allora la richiesta viene gestita da gestore Handler_59 che stampa "Richiesta {} gestita da Handler_59".
- Se il primo intero della lista è maggiore di 9 allora la richiesta viene gestita dal gestore Handler_gt9 che stampa "Messaggio da Handler_gt9: non èstato possibile gestire la richiesta {}. Richiesta modificata". Dopo aver effettuato la stampa Handler_gt9 sottrae al primo intero della lista il secondo intero della lista e lo invia nuovamente ad una nuova catena di gestori.
- Se la richiesta non è una lista di due numeri o il primo intero della lista èminore di 0 la richiesta viene gestita da Default_Handler che stampa semplicemente "Richiesta {} gestita da Default_Handler: non è stato possibile gestire la richiesta {}".
- Nelle suddette stampe la lista nella richiesta deve comparire al posto delle parentesi graffe.

13