



UNIVERSITÀ DEGLI STUDI DI SALERNO

TESINA DESIGN PATTERN

Adapter Pattern

M. Ferraioli D. Tecchia

2015-16

Indice

1	Pattern Strutturali	2
1.1	Adapter Pattern	2
1.1.1	Introduzione e Scopo	2
1.1.2	Partecipanti	2
1.1.3	Class Adapter e Object Adapter	3
1.1.4	Quando è importante usare il Pattern	4
1.1.5	Vantaggi del Pattern	5
2	Esempio di applicazione	6
2.1	Approccio Object Oriented	6
2.1.1	Codice	7
2.2	Approccio "Pythonic way"	8
2.2.1	Codice	8
2.3	Screenshots	9
3	Conclusioni	10

Capitolo 1

Pattern Strutturali

I pattern strutturali consentono di riutilizzare degli oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta alle loro esigenze.

I design patterns strutturali possono essere basati su classi o oggetti: Quelli basati su classi utilizzano l'ereditarietà per generare classi che combinano le proprietà di classi base, mentre, quelli basati su oggetti mostrano come comporre oggetti per realizzare nuove funzionalità.

1.1 Adapter Pattern

1.1.1 Introduzione e Scopo

L'Adapter Pattern, conosciuto anche come Wrapper, ha il fine di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti.

Il problema si presenta ogni qual volta nello sviluppo di un software si devono utilizzare sistemi di supporto (come per esempio librerie) di terze parti la cui interfaccia non è perfettamente compatibile con quella richiesta da software già esistenti senza poter andare a riscrivere parte del sistema. Anche se quest'ultima possibilità fosse possibile sarebbe comunque un compito oneroso.

1.1.2 Partecipanti

L'Adapter pattern ha in totale quattro partecipanti:

Adaptee

Indica un'esistente interfaccia che ha bisogno di essere adattata; esso rappresenta il componente con il quale il client vuole interagire.

Target

Indica l'interfaccia specifica del dominio che utilizza il client; fondamentalmente rappresenta l'interfaccia dell'Adapter che aiuta il client ad interagire con l'Adaptee.

Adapter

Adatta l'interfaccia Adaptee all'interfaccia Target; in altre parole, implementa l'interfaccia di Target, e connette Adaptee con il client.

Client

Il client principale che vuole ottenere l'operazione.

1.1.3 Class Adapter e Object Adapter

Possiamo avere due tipi di Adapter Pattern:

- Class Adapter
- Object Adapter

La principale differenza sta nel fatto che il Class Adapter usa il concetto dell'ereditarietà, mentre l'Object Adapter usa il concetto della composizione.

Class Adapter

Il *Class Adapter*, come accennato, utilizza l'ereditarietà multipla e può contenere una sola classe. Non può contenere un'interfaccia poiché per definizione deve derivare da qualche classe base:

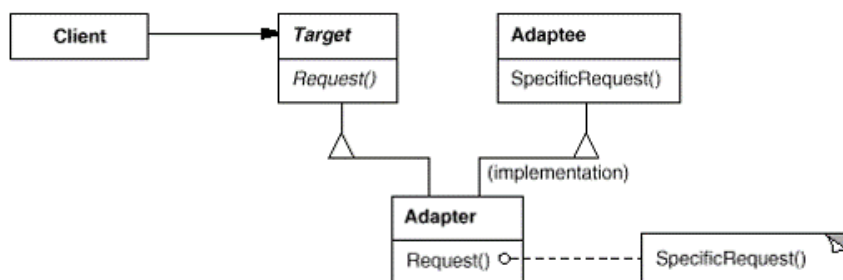


Figura 1.1: UML Class Adapter

Object Adapter

L'*Object Adapter* si basa sulla composizione di oggetti e può contenere interfacce o classi, o entrambe.

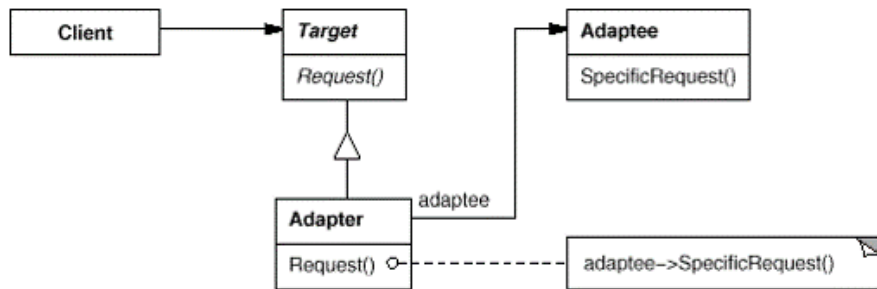


Figura 1.2: UML Object Adapter

1.1.4 Quando è importante usare il Pattern

Possiamo concludere sicuramente, che l'Adapter Pattern risulta un'ottima scelta quando:

- Vogliamo usare classi esistenti, e le loro interfacce non sono compatibili con quelle di cui abbiamo bisogno. Per esempio quando si deve usare del codice di terze parti non pienamente compatibile con il codice del client.
- Si vogliono creare classi riutilizzabili che cooperino con classi impreviste e che non hanno necessariamente interfacce compatibili.
- Si vogliono usare parecchie sottoclassi esistenti, ma risulta poco pratico adattare la loro interfaccia facendo sottoclassi di ognuna. Un Object Adapter può adattare l'interfaccia della sua superclasse. Valido solo per Object Adapter.
- Un Object Adapter può essere utilizzato per un codice che usa classi legacy. Invece, Il Class Adapter viene utilizzato se si ha bisogno dell'ereditarietà singola.

1.1.5 Vantaggi del Pattern

Per quanto riguarda i vantaggi di questo pattern, nel caso del *Class Adapter* abbiamo l'istanziamento di un solo nuovo oggetto, meno codice rispetto all'*Object Adapter*, e la possibilità di fare l'override dei metodi di *Adaptee*; invece, per l'*Object Adapter* possiamo dire che è più flessibile rispetto al *Class Adapter* e non richiede l'impiego di sottoclassi per lavorare. **Adapter** lavora con **Adaptee** e tutte le sue sottoclassi.

Capitolo 2

Esempio di applicazione

Il nostro esempio è molto semplice, abbiamo una piccola interfaccia grafica con un campo di testo dove andremo ad indicare un luogo e, premendo il tasto cerca, verrà eseguita una richiesta HTTP di tipo GET alle API di *openweather.org* per ottenere la temperatura massima, minima e attuale del luogo indicato.

In questo caso le temperature ci verranno fornite in Kelvin, però abbiamo bisogno nel nostro applicativo, di avere il corrispettivo valore in gradi Celsius.

2.1 Approccio Object Oriented

In questo esempio la classe che deve essere adattata è ***Weather*** che ha il metodo ***get-weather***. Il Client, nel nostro caso, si aspetta il metodo ***get-meteo***.

Per "adattare" ***Weather*** abbiamo bisogno di una classe adapter, ***MeteoAdapter***, "l'adattamento", in questo caso, avviene utilizzando il metodo ***get-weather*** all'interno di ***get-meteo***.

Nella seguente pagina andremo ad illustrare il codice python.

2.1.1 Codice

Vediamo il codice dell'esempio:

weather.py

```

1 import requests , json
2
3 class Weather(object):
4
5     __URL__ = 'http://api.openweathermap.org/data/2.5/weather?q='
6     __API_KEY__ = '0cc418d28075d2b9ddc300c3493046e4'
7
8     def get_weather(self , location):
9         r = requests.get(self.__URL__ + location + '&APPID=' +
10                          self.__API_KEY__).json()
11
12         info = {
13             'temp' : r['main']['temp'],
14             'temp_max' : r['main']['temp_max'],
15             'temp_min' : r['main']['temp_min']
16         }
17         return info

```

meteo.py

```

1 class MeteoAdapter(object):
2
3     def __init__(self , legacy_weather):
4         self.legacy_weather = legacy_weather
5
6     def get_meteo(self , posizione):
7         meteo = self.legacy_weather.get_weather(posizione)
8         meteo['temp'] = (meteo['temp'] - 273)
9         meteo['temp_min'] = (meteo['temp_min'] - 273)
10        meteo['temp_max'] = (meteo['temp_max'] - 273)
11        return meteo

```

main.py

```

1 from weather import Weather
2 from meteo import MeteoAdapter
3
4 a_legacy_weather_station = Weather()
5 celsius_weather = MeteoAdapter(a_legacy_weather_station)

```

Come possiamo notare da questo semplice esempio, possiamo identificare *Adapter* con *MeteoAdapter*, *Adaptee* con *Weather*, *Request()* con *get-meteo()* e *SpecificRequest()* con *get-weather()*.

2.2 Approccio "Pythonic way"

Proponiamo anche una variante del primo esempio cercando di sfruttare le potenzialità offerte dal linguaggio Python, andando a sostituire il metodo dell'oggetto "legacy" con il metodo che noi ci aspettiamo.

Come possiamo fare ciò? In Python, sappiamo che ogni oggetto può essere visto come un dizionario accedendo alla proprietà `__dict__` di quest'ultimo. Procediamo con l'inserimento dei metodi richiesti dalla nostra interfaccia nel dizionario dell'oggetto, dove ad ogni metodo da adattare associamo un'espressione lambda che effettua la conversione da Kelvin a Celsius.

2.2.1 Codice

Vediamo il codice dell'esempio:

adapter.py

```
1 class SpecialAdapter(object):
2
3     def __init__(self, legacy_weather, adapted_methods):
4         self.legacy_weather = legacy_weather
5         self.__dict__.update(adapted_methods)
```

weather.py

```
1 import requests, json
2
3 class Weather(object):
4
5     __URL__ = 'http://api.openweathermap.org/data/2.5/weather?q='
6     __API_KEY__ = '0cc418d28075d2b9ddc300c3493046e4'
7
8     def __init__(self, location):
9         self.location = location
10
11     def set_location(self, location):
12         self.location = location
13
14     def get_temp(self):
15         r = requests.get(self.__URL__ + self.location + '&APPID='
16                           + self.__API_KEY__).json()
17         return r['main']['temp']
18
19     def get_temp_min(self):
20         r = requests.get(self.__URL__ + self.location + '&APPID='
21                           + self.__API_KEY__).json()
22         return r['main']['temp_min']
```

```
22     def get_temp_max(self):
23         r = requests.get(self.__URL__ + self.location + '&APPID='
24                           + self.__API_KEY__).json()
25         return r['main']['temp_max']
```

main.py

```
1 kelvin_weather = Weather("")
2 meteo = SpecialAdapter(kelvin_weather, dict(
3     get_temp = lambda: (kelvin_weather.get_temp() - 273),
4     get_temp_min = lambda: (kelvin_weather.get_temp_min() - 273),
5     get_temp_max = lambda: (kelvin_weather.get_temp_max() - 273)
6 ))
```

2.3 Screenshots

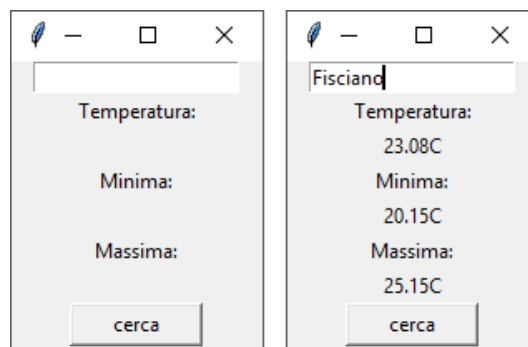


Figura 2.1: Screenshot dell'applicazione

Capitolo 3

Conclusioni

Come abbiamo potuto osservare e come detto in precedenza, l'applicazione del design pattern risulta utile quando abbiamo del codice che non possiamo modificare in maniera diretta e quindi ci risulta comodo adattarlo.

La proprietà `__dict__` ci può tornare utile e, a parer nostro, anche per l'applicazione di altri Design Pattern come il ***Decorator Pattern*** andando a "decorare" il nostro oggetto aggiungendo campi al proprio dizionario e funzionalità al Run-Time.