Università degli Studi di Salerno

Tesina Design Pattern

Adapter Pattern

M. Ferraioli D. Tecchia

2015-16

Indice

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.

I design pattern strutturali basati su classi utilizzano l'ereditarietà per generare classi che combinano le proprietà di classi base.

I design pattern strutturali basati su oggetti mostrano come comporre oggetti per realizzare nuove funzionalità. Dando flessibilità alla composizione che viene modificata a run-time, cosa impossibile da realizzare con le classi.

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 nel progetto di un software si debbano utilizzare sistemi di supporto (come per esempio librerie) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti senza dover andare a riscrivere parte del sistema che risulta 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* usa l'ereditarietà multipla e può contenere solo una classe. Non può contenere un'interfaccia poiché per definizione deve derivare da qualche classe base:

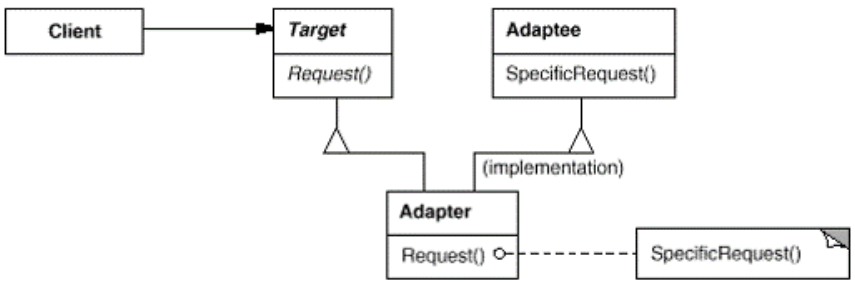

Figura 1.1: UML Class Adapter

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

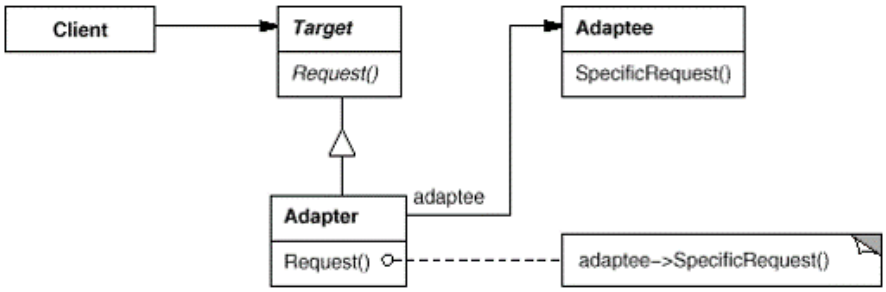

Figura 1.2: UML Object Adapter

1.1.4 Quando è importante usare il Pattern

Detto ciò sicuramente possiamo dire che l'importanza dell'Adapter Pattern possiamo notarla 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.
- Ancora, un Object Adapter può essere utilizzato per un codice che usa classi black-box, o che abbia bisogno dell'ereditarietà multipla; invece, 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'istanzazione di solo un nuovo oggetto, meno codice richiesto rispetto all'Object Adapter, e infine abbiamo 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 sottoclassi per lavorare. **Adapter** lavora con **Adaptee** e tutte le sue sottoclassi.

Capitolo 2

Esempio di appplicazione

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, ma a noi importa averle in Celsius.

2.1 Metodo tradizionale

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

```
import requests, json

class Weather(object):

    __URL__ = 'http://api.openweathermap.org/data/2.5/weather?q='
    __API_KEY__ = '0cc418d28075d2b9ddc300c3493046e4'

    def get_weather(self, location):
        r = requests.get(self.__URL__ + location + '&APPID=' +
            self.__API_KEY__).json()

        info = {
            'temp'       : r['main']['temp'],
            'temp_max'   : r['main']['temp_max'],
            'temp_min'   : r['main']['temp_min']
        }
        return info
```

meteo.py

```
class MeteoAdapter(object):

    def __init__(self, legacy_weather):
        self.legacy_weather = legacy_weather

    def get_meteo(self, posizione):
        meteo = self.legacy_weather.get_weather(posizione)
        meteo['temp']       = (meteo['temp'] - 273)
        meteo['temp_min']   = (meteo['temp_min'] - 273)
        meteo['temp_max']   = (meteo['temp_max'] - 273)
        return meteo
```

main.py

```
from weather import Weather
from meteo import MeteoAdapter

a_legacy_weather_station = Weather()
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 Metodo "Pythonic way"

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

Come possiamo fare ciò? In Python, ogni oggetto può essere visto come un dizionario accedendo alla proprietà **__dict__**.

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='
               + self.__API_KEY__).json()
16         return r['main']['temp']
17
18     def get_temp_min(self):
19         r = requests.get(self.__URL__ + self.location + '&APPID='
               + self.__API_KEY__).json()
20         return r['main']['temp_min']
21
22     def get_temp_max(self):
23         r = requests.get(self.__URL__ + self.location + '&APPID='
               + self.__API_KEY__).json()
24         return r['main']['temp_max']
```

main.py

```
kelvin_weather = Weather("fisciano")
meteo = SpecialAdapter(kelvin_weather, dict(
    get_temp = lambda: (kelvin_weather.get_temp() - 273),
    get_temp_min = lambda: (kelvin_weather.get_temp_min() - 273),
    get_temp_max = lambda: (kelvin_weather.get_temp_max() - 273)
))
```

Capitolo 3

Conclusioni

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

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