

Programmazione Avanzata

Design Pattern: Adapter

Programmazione Avanzata a.a. 2023-24
A. De Bonis

97

Adapter

- L'Adapter è un design pattern strutturale che ci aiuta a rendere compatibili interfacce tra di loro incompatibili.
- In altre parole l'Adapter crea un livello che permette di comunicare a due interfacce differenti che non sono in grado di comunicare tra di loro
- **Esempio:** Un sistema di e-commerce contiene una funzione `calculate_total(order)` in grado di calcolare l'ammontare di un ordine solo in Corone Danesi (DKK).
 - Vogliamo aggiungere il supporto per valute di uso più comune quali i Dollari USA (USD) e gli Euro (EUR).
 - Se possediamo il codice sorgente del sistema possiamo estenderlo in modo da incorporare nuove funzioni per effettuare le conversioni da DKK a EUR e USD.
 - Che accade però se non disponiamo del sorgente perché l'applicazione ci è fornita da una libreria esterna? In questo caso, possiamo usare la libreria ma non modificarla o estenderla.
 - La soluzione fornita dall'Adapter consiste nel creare un livello extra (wrapper) che effettua la conversione tra i formati delle valute.

Programmazione Avanzata a.a. 2023-24
A. De Bonis

98

Adapter

- L'adapter in generale è utile quando vogliamo usare un'interfaccia che ci aspettiamo fornisca una certa funzione f() ma disponiamo solo della funzione g().
 - L'adapter può essere usato per convertire la nostra funzione g() nella funzione f().
 - La conversione potrebbe riguardare anche il numero di parametri. Supponiamo, ad esempio, di voler usare un'interfaccia con una funzione che richiede tre parametri ma abbiamo una funzione che prende due parametri.

Programmazione Avanzata a.a. 2023-24
A. De Bonis

99

Adapter: un semplice esempio

- La nostra applicazione ha una classe Computer che mostra l'informazione di base riguardo ad un computer.

```
class Computer:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'the {} computer'.format(self.name)
    def execute(self): return 'executes a program'
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

100

Adapter: un semplice esempio

- Decidiamo di arricchire la nostra applicazione con altre funzionalità e per nostra fortuna scopriamo due classi che potrebbero fare al nostro caso in due distinte librerie: la classe Synthesizer e la classe Human.

```
class Synthesizer:
    def __init__(self, name):
        self.name = name
    def __str__(self): return 'the {} synthesizer'.format(self.name)
    def play(self): return 'is playing an electronic song'

class Human:
    def __init__(self, name): self.name = name
    def __str__(self):
        return '{} the human'.format(self.name)
    def speak(self):
        return 'says hello'
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

101

Adapter: un semplice esempio

- Poniamo le due classi in un modulo separato.
- Problema: il client sa solo che può invocare il metodo execute() e non ha alcuna idea dei metodi play() o speak().
- Come possiamo far funzionare il codice senza modificare le classi Synthesizer e Human?
- Soluzione : design pattern adapter.

Programmazione Avanzata a.a. 2023-24
A. De Bonis

102

Adapter: un'implementazione che usa un dizionario di metodi

- Creiamo una classe generica Adapter che ci permetta di unificare oggetti di diverse interfacce
- Un'istanza della classe Adapter ha una variabile obj che è un'istanza di una delle classi che vogliamo includere nella nostra applicazione, ad esempio un'istanza di Human.
- Il metodo `__init__` di Adapter inserisce in `__dict__` dell'istanza self alcune coppie chiave/valore per associare a ciascun metodo dell'interfaccia che vogliamo usare il metodo corrispondente della classe di obj, ad esempio si può associare il metodo `execute` al metodo `Human.speak`. Nel nostro esempio c'è un solo metodo (`execute`) nell'interfaccia che vogliamo utilizzare

Programmazione Avanzata a.a. 2023-24
A. De Bonis

103

Adapter: un'implementazione che usa un dizionario di metodi

- L'argomento obj del metodo `__init__()` è l'oggetto che vogliamo adattare
- `adapted_methods` è un dizionario che contiene le coppie chiave/valore dove la chiave è il metodo che il client invoca e il valore è il metodo della libreria che dovrebbe essere invocato.

```
class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)
    def __str__(self):
        return str(self.obj)
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

104

Adapter: un'implementazione che usa un dizionario di metodi

objects è la lista di tutti gli oggetti. L'istanza di computer viene aggiunta alla lista senza adattamenti. Gli oggetti incompatibili (istanze di Human o Synthesizer) sono prima adattate usando la classe Adapter. Il client può usare execute() su tutti gli oggetti senza essere a conoscenza delle differenze tra le classi usate.

```
def main():
    objects = [Computer('Asus')]
    synth = Synthesizer('moog')
    objects.append(Adapter(synth, dict(execute=synth.play) ))
    human = Human('Bob')
    objects.append(Adapter(human, dict(execute=human.speak)))
    for i in objects:
        print('{} {}'.format(str(i), i.execute()))
if __name__ == "__main__": main()
```

the Asus computer executes a program
the moog synthesizer is playing an electronic song
Bob the human says hello

Programmazione Avanzata a.a. 2023-24
A. De Bonis

105

Adapter: un'implementazione che usa l'ereditarietà

La classe Adapter estende la classe che vogliamo utilizzare sovrascrivendo i metodi dell'interfaccia usati dall'applicazione in modo che invocano quelli di WhatIHave.

```
class WhatIHave:
    `interfaccia a nostra disposizione`
    def g(self): pass
    def h(self): pass

class WhatIWant:
    `interfaccia che vogliamo usare`
    def f(self): pass

class Adapter(WhatIWant):
    `adatta WhatIHave a WhatIWant`
    def __init__(self, whatIHave):
        self.whatIHave = whatIHave
    def f(self):
        self.whatIHave.g()
        self.whatIHave.h()
```

```
class WhatIUse:
    def op(self, whatIWant):
        `metodo dell'applicazione che usa f`
        whatIWant.f()

whatIUse = WhatIUse()
whatIHave = WhatIHave()
adapt= Adapter(whatIHave)

#op() riceve un'istanza di Adapter che ha gli stessi metodi
#dell'interfaccia desiderata WhatIWant, cioe` il metodo f()
#che viene invocato all'interno di op()

whatIUse.op(adapt)
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

106

Adapter: un'implementazione che usa l'ereditarietà– esempio Computer

```
class Human:
    ...
class Synthesizer:
    ...
class Computer:
    ...
class Adapter(Computer):
    def __init__(self, wih):
        self.wih=wih
    def execute(self):
        if isinstance(self.wih, Synthesizer):
            return self.wih.play()
        if isinstance(self.wih, Human):
            return self.wih.speak()
```

```
class WhatIUse:
    def op(self, comp):
        return comp.execute()
```

```
whatIUse = WhatIUse()
human = Human('Bob')
adapt= Adapter(human)
```

```
#op() riceve un'istanza di Adapter il cui
#metodo execute() si comporta come
#speak() della classe Human
print(whatIUse.op(adapt))
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

107

Adapter

- Consideriamo il seguente esempio:
- Immaginiamo di avere una classe Page che puo` essere usata per produrre una pagina a partire dal titolo, dai paragrafi del testo e utilizzando un oggetto che fornisce l'interfaccia del renderer.

```
class Page:
    def __init__(self, title, renderer):
        if not isinstance(renderer, Renderer):
            raise TypeError("Expected object of type Renderer, got {}".
                             format(type(renderer).__name__))
        self.title = title
        self.renderer = renderer
        self.paragraphs = []

    def add_paragraph(self, paragraph):
        self.paragraphs.append(paragraph)

    def render(self):
        self.renderer.header(self.title)
        for paragraph in self.paragraphs:
            self.renderer.paragraph(paragraph)
        self.renderer.footer()
```

A. De Bonis

108

Adapter

- La classe Page ha bisogno di usare i metodi dell'interfaccia del renderer e cioè i metodi `header(str)`, `paragraph(str)` e `footer()`.

- `__init__` verifica che l'argomento `renderer` sia di tipo `Renderer`
- il metodo `render` costruisce la pagina invocando i metodi dell'interfaccia `renderer`.

```
class Page:
    def __init__(self, title, renderer):
        if not isinstance(renderer, Renderer):
            raise TypeError("Expected object of type Renderer, got {}".format(type(renderer).__name__))
        self.title = title
        self.renderer = renderer
        self.paragraphs = []

    def add_paragraph(self, paragraph):
        self.paragraphs.append(paragraph)

    def render(self):
        self.renderer.header(self.title)
        for paragraph in self.paragraphs:
            self.renderer.paragraph(paragraph)
        self.renderer.footer()
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

109

Adapter

- Immaginiamo che siano disponibili due classi `TextRender` e `HtmlWriter` che potrebbero essere usate da `Page` per costruire pagine testuali e pagine html, rispettivamente
- Supponiamo che `TextRender` supporti l'interfaccia `renderer`, cioè fornisca i tre metodi `header(str)`, `paragraph(str)` e `footer()`.
 - Un'istanza di `TextRender` può essere passata al costruttore di `Page`
- Supponiamo invece che `HtmlWriter` fornisca solo i due metodi `header(str)` e `footer()` dell'interfaccia `renderer` e che questi però non facciano ciò che ci si aspetterebbe.
 - Un'istanza di `HtmlWriter` **non** può essere passata al costruttore di `Page`
- Si potrebbe allora pensare di estendere `HtmlWriter` e di aggiungere alla classe derivata il metodo `paragraph(str)` e di reimplementare i metodi `header(str)` e `footer()` sovrascrivendoli.
 - In questo modo però la nuova classe conterrebbe sia metodi dell'interfaccia `renderer` sia quelli della classe base `HtmlWriter`

Programmazione Avanzata a.a. 2023-24
A. De Bonis

110

Adapter

- Un approccio alternativo consiste nel costruire una classe che funge da adapter e che ha al suo interno una variabile di istanza della classe `htmlWriter`. I metodi di `HtmlRenderer` vengono implementati invocando quelli di `htmlWriter`.

```
class HtmlRenderer:
    def __init__(self, htmlWriter):
        self.htmlWriter = htmlWriter

    def header(self, title):
        self.htmlWriter.header()
        self.htmlWriter.title(title)
        self.htmlWriter.start_body()

    def paragraph(self, text):
        self.htmlWriter.body(text)

    def footer(self):
        self.htmlWriter.end_body()
        self.htmlWriter.footer()
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

111

Adapter

- Esempio dell'uso della classe `Page` con `TextRender` e `HtmlRender`. Il costruttore di `TextRenderer` prende in input il numero di caratteri che definisce l'ampiezza del testo mentre quello di `HtmlRenderer` prende in input un'istanza di `HtmlWriter`. Il costruttore di `HtmlWriter` prende in input un file object.

```
textPage = Page(title, TextRenderer(22))
textPage.add_paragraph(paragraph1)
textPage.add_paragraph(paragraph2)
textPage.render()

htmlPage = Page(title, HtmlRenderer(HtmlWriter(file)))
htmlPage.add_paragraph(paragraph1)
htmlPage.add_paragraph(paragraph2)
htmlPage.render()
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

112

Adapter: miglioramenti alla classe Page

- La classe Page non ha bisogno di conoscere come sia fatta la classe Renderer ma è solo interessato al fatto che fornisca l'interfaccia renderer e cioè i metodi header(str), paragraph(str) e footer().
- Per essere sicuri che l'argomento renderer passato ad __init__ sia un'istanza di Renderer, al posto dello statement **if not isinstance(...)**, si potrebbe usare **assert isinstance(renderer, Renderer)**. Questo approccio però pone due problemi:
 - Lancia AssertionError piuttosto che la più specifica TypeError.
 - Se l'utente esegue il programma con l'opzione -O, l'assert viene ignorato e in seguito il metodo render() lancerà un AttributeError.
- Per questo motivo il codice di __init__ usa lo statement **if not isinstance(...)**
- Potrebbe sembrare che questo approccio ci costringa ad usare esclusivamente renderer che sono istanze di sottoclassi di una stessa classe base Renderer.
- Rispetto a linguaggi OOP quali C++, Python ci consente di usare un approccio alternativo basato sul modulo abc (abstract base class): l'idea è di creare oggetti che forniscano una particolare interfaccia ma che non debbano necessariamente essere sottoclassi di una particolare classe base (duck typing).

Programmazione Avanzata a.a. 2023-24
A. De Bonis

113

Adapter

- La classe Renderer reimplementa il metodo speciale __subclasshook__(). Questo metodo viene utilizzato dalla funzione built-in isinstance()
- Il metodo __subclasshook__() è un metodo di classe e per prima cosa controlla se la classe su cui è invocato è Renderer e in caso contrario lancia l'eccezione NotImplemented
 - in questo modo il comportamento di __subclasshook__ non viene ereditato dalle eventuali sottoclassi e le sottoclassi possono eventualmente aggiungere nuovi criteri alla classe astratta. Ovviamente possiamo fare in modo che __subclasshook__() di una sottoclasse invochi Renderer.__subclasshook__() esplicitamente se vogliamo che ne erediti il comportamento.

```
class Renderer(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(Class, Subclass):
        if Class is Renderer:
            attributes = collections.ChainMap(*(Superclass.__dict__
                                                for Superclass in Subclass.__mro__)
            methods = ("header", "paragraph", "footer")
            if all(method in attributes for method in methods):
                return True
            return NotImplemented
```

114

Il metodo `__subclasshook__`

- `__subclasshook__(subclass)` può essere sovrascritto in una abstract base class.
 - deve essere ridefinito come metodo di classe.


```
@classmethod
def __subclasshook__(cls, subclass):.....
```
 - controlla se subclass è considerata una sottoclasse della classe ABC in cui il metodo è sovrascritto (rappresentata dal parametro `cls`).
 - In questo modo è possibile modificare il comportamento di `issubclass` e di `isinstance` senza bisogno di invocare `register()` su ogni classe che vogliamo venga considerata sottoclasse dell'ABC che stiamo implementando.
- `__subclasshook__` dovrebbe restituire `True`, `False` o `NotImplemented`. Se restituisce `True`, `subclass` viene considerata sottoclasse dell'ABC. Se restituisce `False`, `subclass` non è considerata sottoclasse dell'ABC. Se restituisce `NotImplemented`, il controllo sulla sottoclasse continua con il meccanismo usuale.

Programmazione Avanzata a.a. 2023-24
A. De Bonis

115

Adapter

- se `Class` è la classe `Render` allora
 - viene creata una `ChainMap` (si veda slide successiva) dei `__dict__` di tutte le classi presenti nello `__mro__` di `Subclass`.
 - viene creata una tupla **methods** dei metodi che devono essere controllati
 - viene restituito `True` se tutti i metodi in **methods** sono presenti in `Subclass` o in una delle sue superclassi. Se vogliamo essere certi di non confondere una proprietà di una delle classi con il metodo di subclass da controllare, dobbiamo verificare che `method` sia callable.

`all(iterable)` restituisce `True` se tutti gli elementi in `iterable` sono `True` o se `iterable` è vuoto.

```
class Renderer(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(Class, Subclass):
        if Class is Renderer:
            attributes = collections.ChainMap(*(Superclass.__dict__
                                                for Superclass in Subclass.__mro__))
            methods = ("header", "paragraph", "footer")
            if all(method in attributes for method in methods):
                return True
            return NotImplemented
```

Programmazione Avanzata a.a. 2023-24
A. De Bonis

116

La classe ChainMap

- ChainMap è una classe del modulo collections e serve a creare una singola view che raggruppa più mapping (istanze di dict o di altri tipi di mapping).
- `class collections.ChainMap(*maps)` crea la view a partire dai mapping in maps. Se maps non viene fornito allora viene creato un singolo dizionario vuoto in modo che una ChainMap abbia almeno un mapping.
- I mapping sottostanti sono memorizzati in una lista a cui si può accedere attraverso l'attributo maps.
- La ricerca in una ChainMap avviene effettuando la ricerca in tutti i mapping sottostanti fino a che non viene trovata una chiave. Le operazioni di scrittura e aggiornamento invece vengono effettuate solo sul primo mapping.
- Le modifiche apportate ai mapping sottostanti sono visibili nella ChainMap.

Programmazione Avanzata a.a. 2023-24
A. De Bonis

117

Adapter

- Sovrascrivere `__subclasshook__()` in `Renderer` è molto utile ma scrivere linee di codice così complesse ogni volta che occorre fornire un meccanismo per controllare un'interfaccia, comporta una duplicazione di codice che è bene evitare.
- Le linee di codice differirebbero infatti molto poco: la classe base e i metodi supportati.
- Per evitare questa duplicazione di codice, implementiamo un decorator factory che restituisce un decoratore di classe che dota la classe della definizione di `__subclasshook__` di cui abbiamo bisogno.

```
def has_methods(*methods):
    def decorator(Base):
        def __subclasshook__(Class, Subclass):
            if Class is Base:
                attributes = collections.ChainMap(*(Superclass.__dict__
                                                    for Superclass in Subclass.__mro__))
                if all(method in attributes for method in methods):
                    return True
                return NotImplemented
            Base.__subclasshook__ = classmethod(__subclasshook__)
            return Base
        return decorator
```

uso del decorator factory per decorare `Renderer`. Nell'implementazione del libro il decorator factory si trova in `Qtrac`.

```
@Qtrac.has_methods("header", "paragraph", "footer")
class Renderer(metaclass=abc.ABCMeta): pass
```

A. De Bonis

118