Programmazione Avanzata

Design Pattern: adapter

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 perche' 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 la conversione tra i formati delle valute.

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 puo` 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.

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'
```

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'
```


77

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.

78

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 puo` associare il metodo execute al metodo Human.speak. Nel nostro esempio c'è un solo metodo (execute) nell'interfaccia che vogliamo utiliizzare

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)
```

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 differenza 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()
```

```
>>> python3 adapter.py
the Asus computer executes a program
the moog synthesizer is playing an electronic song
Bob the human says hello
```


81

Adapter: un'implementazione che usa l'ereditarietà (1)

La classe Adapter estende la classe che vogliamo utilizzare sovrascrivendo i metodi dell'interfaccia usati dall'applicazione.

```
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)
```


82

Adapter: un'implementazione che usa l'ereditarietà (1) – 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))
```


83

Adapter: un'altra implementazione che usa l'ereditarietà (2)

A differenza di quella precedente, in questa implementazione il metodo op invocato dall'applicazione è quello di WhatIUse2 che prende in input un oggetto WhatIHave e invoca f() sulla sua versione ''adattata'' .

```
class WhatIHave:
        def g(self): pass
        def h(self): pass

class WhatIWant:
        def f(self): pass

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

```
class WhatIUse:
        def op(self, whatIWant):
                whatIWant.f()


# costruisce l'adapter in op():
class WhatIUse2(WhatIUse):
        def op(self, whatIHave):
                Adapter(whatIhave).f()

#non c'è bisono di creare un'istanza di Adapter
#perche' op() di WhatIUse2 riceve un'istanza di
#WhatIHave:
whatIUse2 = WhatIUse2()
whatIHave = WhatIHave()
whatIUse2.op(whatIHave)
```

prime 4 classi stesse di prima

84

Adapter: un'implementazione che usa l'ereditarietà (2) - 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()

class WhatIUse2(WhatIUse):
        def op(self, wih):
                if not isinstance(wih,Computer):
                        wih=Adapter(wih)
                return super().op(wih)

whatIUse2 = WhatIUse2()
human = Human('Bob')
print(whatIUse2.op(human))
computer =Computer('Asus')
print(whatIUse2.op(computer))
```


Adapter: un'implementazione che usa l'ereditarietà (2bis)- 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()

class WhatIUse2(WhatIUse):
        def op(self, wih):
                Adapter(wih).op(wih)

whatIUse2 = WhatIUse2()
whatIUse= WhatIUse()
human = Human('Bob')
print(whatIUse2.op(human))
computer =Computer('Asus')
print(whatIUse.op(computer))
```

Adapter: ancora un'altra implementazione che usa l'ereditarietà (3)

La classe WhatIHave2 unifica le due interfacce e adatta le funzionalita` di WhatIHave a WhatIWant

```
class WhatIHave:
        def g(self): pass
        def h(self): pass


class WhatIWant:
        def f(self): pass
```

```
class WhatIUse:
        def op(self, whatIWant):
                whatIWant.f()

# WhatIHave2 effettua l'adattamento
# → una classe per ogni classe da adattare :-(
class WhatIHave2(WhatIHave, WhatIWant):
        def f(self):
                self.g()
                self.h()


whatIUse = WhatIUse()
whatIhave2=WhatIHave2()
whatIUse.op(whatIHave2)
```

prime 3 classi stesse di prima

87

Adapter: un'implementazione che usa l'ereditarietà (3) - esempio Computer

```
class Human:
        ...
class Synthesizer:
        ...
class Computer:
        ...
```

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

class Human2(Human, Computer):
        def execute(self):
                return self.speak()


class Synthesizer2(Synthesizer, Computer):
        def execute(self):
                return self.play()

whatIUse = WhatIUse()
human2 = Human2('Bob')
print(whatIUse.op(human2))
computer=Computer('Asus')
print(whatIuse.op(computer))
```


88

Adapter: ancora un'altra implementazione che usa l'ereditarietà (4)

Questa implementazione estende WhatIHave con una classe WhatIHave3 che ha una classe interna che funge da adapter. Le istanze di questo adapter hanno una variabile outer di tipo WhatIHave3. Il metodo f() dell'adapter invoca i metodi g() e h() di WhatIHave su outer. WhatIHave3 ha un metodo di istanza whatIWant che restituisce un'istanza dell'adapter in cui outer è l'oggetto su cui whatIWhant è invocato.

```
class WhatIHave:
        def g(self): pass
        def h(self): pass

class WhatIWant:
        def f(self): pass


class WhatIUse:
        def op(self, whatIWant):
                whatIWant.f()
```

prime 3 classi stesse di prima

```
# adapter interno a WhatIHave3
# → una classe per ogni classe da adattare :-(
class WhatIHave3(WhatIHave):
        class InnerAdapter(WhatIWant):
                def __init__(self, outer):
                        self.outer = outer
                def f(self):
                        self.outer.g()
                        self.outer.h()
        def whatIWant(self):
                return WhatIHave3.InnerAdapter(self)

whatIUse = WhatIUse()
whatIHave3=WhatIHave3()
whatIUse.op(whatIHave3.whatIWant())
```


Adapter: un'implementazione che usa l'ereditarietà (4)- esempio Computer

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

```
class Human3(Human):
    #adattatore interno
    class InnerAdapter(Computer):
        def __init__(self, outer):
            self.outer = outer
        def execute(self):
            return self.outer.speak()
    def whatIWant(self):
        return Human3.InnerAdapter(self)

class Synthesizer3(Synthesizer):
        ...

whatIUse = WhatIUse()
human3=Human3('Bob')
print(whatIUse.op(human3.whatIWant()))
```

Adapter: un'implementazione che usa l'ereditarietà (5)- esempio Computer

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

```
#funzione che restituisce una classe derivata da c (classe WhatIhave)
def createClass(c):
        class hs(c):
                def execute(self):
                        if isinstance(self,Human) :
                                return self.speak()
                        if isinstance(self,Synthesizer) :
                                return self.play()

        return hs

whatIUse = WhatIUse()
myclass=createClass(Human) #myclass estende Human e Computer
human = myclass('Bob') #instanza di myclass di nome Bob
print(whatIUse.op(human)) #invoca human.speak()
computer = Computer('Asus')
print(whatIUse.op(computer))
```

Esercizio: posso rendere createClass un decoratore di classe? Se sì , modificate il codice.

91

Design Pattern Proxy

- Proxy è un design pattern strutturale
  - fornisce una classe surrogato che nasconde la classe che svolge effettivamente il lavoro
- Quando si invoca un metodo del surrogato, di fatto viene utilizzato il metodo della classe che lo implementa.
- Proxy è un caso particolare del design pattern state
- Quando un oggetto surrogato è creato, viene fornita un'implementazione alla quale vengono inviate tutte le chiamate dei metodi

92

Design Pattern Proxy

Usi di Proxy :

1. **Remote proxy** è un proxy per un oggetto in un diverso spazio di indirizzi.
   - Il libro "Python in Practice" descrive nel capitolo 6 la libreria RPyC (Remote Python Call) che permette di creare oggetti su un server e di avere proxy di questi oggetti su uno o più client

2. **Virtual proxy** è un proxy che fornisce una "lazy initialization" per creare oggetti costosi su richiesta solo se se sono realmente necessari.

3. **Protection proxy** è un proxy usato quando vogliamo che il programmatore lato client non abbia pieno accesso all'oggetto.

4. **Smart reference** è un proxy usato per aggiungere azioni aggiuntive quando si accede all'oggetto. Per esempio, per mantenere traccia del numero di riferimenti ad un certo oggetto

Design Pattern Proxy

```
class Implementation:
        def f(self):
                print("Implementation.f()")
        def g(self):
                print("Implementation.g()")
        def h(self):
                print("Implementation.h()")
class Proxy:
        def __init__(self):
                self.__implementation = Implementation()
        # Passa le chiamate ai metodi all'implementazione:
        def f(self): self.__implementation.f()
        def g(self): self.__implementation.g()
        def h(self): self.__implementation.h()
p = Proxy()
p.f(); p.g(); p.h()
```

Design Pattern Proxy

- Non è necessario che **Implementation** abbia la stessa interfaccia di **Proxy** ma è comunque conveniente avere un'interfaccia comune in modo che **Implementation** sia forzata a fornire tutti i metodi che **Proxy** ha bisogno di invocare.

Design Pattern Proxy

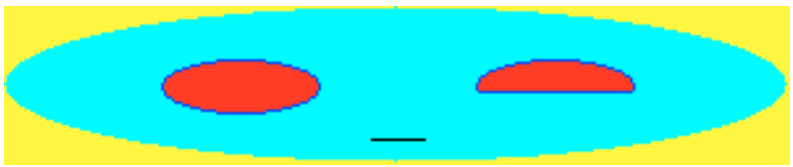
L'uso di **__getattr__( )** rende **Proxy2** completamente generica e non legata ad una particolare implementazione

```
class Implementation2:
        def f(self):
                print("Implementation.f()")
        def g(self):
                print("Implementation.g()")
        def h(self):
                print("Implementation.h()")
class Proxy2:
        def __init__(self):
                self.__implementation = Implementation2()
        def __getattr__(self, name):
                return getattr(self.__implementation, name)
p = Proxy2()
p.f(); p.g(); p.h();
```

Design Pattern Proxy: esempio

- Abbiamo bisogno di creare immagini delle quali però una sola verrà usata realmente alla fine.
- Abbiamo un modulo Image e un modulo quasi equivalente più veloce cyImage. Entrambi i moduli creano le loro immagini in memoria.
- Siccome avremo bisogno solo di un'immagine tra quelle create, sarebbe meglio utilizzare dei proxy "leggeri" che permettano di creare una vera immagine solo quando sapremo di quale immagine avremo bisogno.
- L'interfaccia Image.Image consiste di 10 metodi in aggiunta al costruttore: load(), save(), pixel(), set_pixel(), line(), rectangle(), ellipse(), size(), subsample(), scale().
  - Non sono elencati alcuni metodi statici aggiuntivi, quali Image.Image.color_for_name() e Image.color_for_name().

Design Pattern Proxy: esempio



```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
    for color in ("yellow", "cyan", "blue", "red", "black"))
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```

Design Pattern Proxy: esempio

- La classe ImageProxy può essere usata al posto di Image.Image (o di qualsiasi altra classe immagine che supporta l'interfaccia Image) a patto che l'interfaccia incompleta fornita da ImageProxy sia sufficiente.
- Un oggetto ImageProxy non salva un'immagine ma mantiene una lista di tuple di comandi dove il primo elemento in ciascuna tupla è una funzione o un metodo unbound (non legato ad una particolare istanza) e i rimanenti elementi sono gli argomenti da passare quando la funzione o il metodo è invocato.

```
class ImageProxy:

    def __init__(self, ImageClass, width=None, height=None, filename=None):
        assert (width is not None and height is not None) or \
                filename is not None
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
        else:
            self.commands = [(self.Image, width, height)]

    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```


Design Pattern Proxy: esempio

Quando viene creato un ImageProxy, gli deve essere fornita l'altezza e la larghezza dell'immagine o il nome di un file.
Se viene fornito il nome di un file, l'ImageProxy immagazzina una tupla con il costruttore Image.Image(), None e None (per la larghezza e l'altezza) e il nome del file da cui il metodo load di ImageClass carichera` le informazioni per costruire l'immagine.
Se non viene fornito il nome di un file allora viene immagazzinato il costruttore Image.Image() insieme alla larghezza e l'altezza.

```
class ImageProxy:

    def __init__(self, ImageClass, width=None, height=None, filename=None):
        assert (width is not None and height is not None) or \
                filename is not None
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
        else:
            self.commands = [(self.Image, width, height)]

    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```

Design Pattern Proxy: esempio

- La classe Image.Image ha 4 metodi: line(), rectangle(), ellipse(), set_pixel().
- La classe ImageProxy supporta pienamente questa interfaccia solo che invece di eseguire questi comandi, semplicemente li aggiunge insieme ai loro argomenti alla lista.
- Il metodo inserito all'inizio della tupla è unbound in quanto non è legato ad un'istanza di self.Image (self.Image è la classe che fornisce il metodo)

```
def set_pixel(self, x, y, color):
    self.commands.append((self.Image.set_pixel, x, y, color))

def line(self, x0, y0, x1, y1, color):
    self.commands.append((self.Image.line, x0, y0, x1, y1, color))

def rectangle(self, x0, y0, x1, y1, outline=None, fill=None):
    self.commands.append((self.Image.rectangle, x0, y0, x1, y1,
            outline, fill))

def ellipse(self, x0, y0, x1, y1, outline=None, fill=None):
    self.commands.append((self.Image.ellipse, x0, y0, x1, y1,
            outline, fill))
```


Design Pattern Proxy: esempio

- Solo quando si sceglie di salvare l'immagine, essa viene effettivamente creata e viene quindi pagato il prezzo relativo alla sua creazione, in termini di computazione e uso di memoria.
- Il primo comando della lista self.commands è sempre quello che crea una nuova immagine. Quindi il primo comando viene trattato in modo speciale salvando il suo valore di ritorno (che è un Image.Image o un cyImage.Image) in image.
- Poi vengono invocati nel for i restanti comandi passando image come argomento insieme agli altri argomenti.
- Alla fine, si salva l'immagine con il metodo Image.Image.save().

```
def save(self, filename=None):
    command = self.commands.pop(0)
    function, *args = command
    image = function(*args)
    for command in self.commands:
        function, *args = command
        function(image, *args)
    image.save(filename)
    return image
```

Design Pattern Proxy: esempio

- Il metodo Image.Image.save() non ha un valore di ritorno (sebbene possa lanciare un'eccezione se accade un errore).
- L'interfaccia è stata modificata leggermente per ImageProxy per consentire a save() di restituire l'immagine Image.Image creata per eventuali ulteriori usi dell'immagine.
- Si tratta di una modifica innocua in quanto se il valore di ritorno è ignorato, esso viene scartato.

```
    def save(self, filename=None):
        command = self.commands.pop(0)
        function, *args = command
        image = function(*args)
        for command in self.commands:
            function, *args = command
            function(image, *args)
        image.save(filename)
        return image
```


Design Pattern Proxy: esempio

- Se un metodo non supportato viene invocato (ad esempio, pixel(()), Python lancia un AttributeError.
- Un approccio alternativo per gestire i metodi che non possono essere delegati è di creare una vera immagine non appena uno di questi metodi è invocato e da quel momento usare la vera immagine.

Design Pattern Proxy: esempio

Questo codice prima crea alcune costanti colore con la funzione color_for_name del modulo Image e poi crea un oggetto ImageProxy passando come argomento a __init__ la classe che si vuole usare. L'ImageProxy creato è usato quindi per disegnare e infine salvare l'immagine risultante.

```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
    for color in ("yellow", "cyan", "blue", "red", "black"))
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```


105

Design Pattern Proxy: esempio

- Il codice alla pagina precedente avrebbe funzionato allo stesso modo se avessimo usato Image.image() al posto di ImageProxy().
- Usando un image proxy, la vera immagine non viene creata fino a che il metodo save non viene invocato. In questo modo il costo per creare un'immagine prima di salvarlo è estremamente basso (sia in termini di memoria che di computazione) e se alla fine scartiamo l'immagine senza salvarla perdiamo veramente poco.
- Se usassimo Image.Image, verrebbe effettivamente creato un array di dimensioni width × height di colori e si farebbe un costoso lavoro di elaborazione per disegnare (ad esempio, per settare ogni pixel del rettangolo) che verrebbe sprecato se alla fine scartassimo l'immagine.

106