Programmazione Avanzata

Design Pattern: Mediator

31

Il Design Pattern Mediator

- Il Design Pattern Mediator e` un design pattern comportamentale che fornisce un mezzo per creare un oggetto che incapsula le interazioni tra altri oggetti.
- Cio` consente di stabilire relazioni tra oggetti senza che questi abbiano conoscenza diretta l'uno dell'altro.
- Per esempio se si verifica un evento che richiede l'attenzione di alcuni oggetti, tale evento sara` comunicato al mediatore che mandera` una notifica agli oggetti interessati
- Il design pattern mediator evita il tight coupling (forte dipendenza tra un gruppo di oggetti)
  → rende possibile cambiare l'interazione tra gli oggetti senza dover apportare modifiche agli oggetti stessi
  →facilita l'implementazione, il testing, la riusabilita` degli oggetti.

32

Il Design Pattern Mediator: un esempio

- Vogliamo creare delle form contenenti text widget e button widget
- Cio` e` di grande utilita` nella programmazione GUI.
- L'interazione tra i widget della form sara` gestita da un mediator
- La classe Form fornisce i metodi create_widgets() e create_mediator()

```
class Form:

    def __init__(self):
        self.create_widgets()
        self.create_mediator()
```


Il Design Pattern Mediator: un esempio

- La form ha
  - due widget per inserire testo: una per il nome dell'utente, l'altra per l'indirizzo email
  - due bottoni: OK e CANCEL

```
    def create_widgets(self):
        self.nameText = Text()
        self.emailText = Text()
        self.okButton = Button("OK")
        self.cancelButton = Button("Cancel")
```

Il Design Pattern Mediator: un esempio

- **Ogni form ha un singolo mediator associato**.
- Il metodo __init__() di Mediator riceve come argomenti una o piu` coppie (widget, callable), ciascuna delle quali descrive una relazione che il mediatore deve supportare.
- Nel codice riportato di seguito, le coppie passate al mediatore fanno in modo che se cambia il testo di uno dei widget per l'inserimento di testo allora viene invocato il metodo Form.update_ui(); mentre se viene cliccato uno dei bottoni allora viene invocato il metodo Form.clicked().
- Dopo aver creato il mediatore, viene invocato il metodo update_ui() per inizializzare la form.

```
def create_mediator(self):
    self.mediator = Mediator(((self.nameText, self.update_ui),
            (self.emailText, self.update_ui),
            (self.okButton, self.clicked),
            (self.cancelButton, self.clicked)))
    self.update_ui()
```


35

Il Design Pattern Mediator: un esempio

- Questo metodo abilita il bottone OK se entrambi i widget per inserire testo contengono del testo; altrimenti disabilita il bottone.
- Questo metodo viene invocato ogni volta che cambia il testo in uno dei due widget.

```
def update_ui(self, widget=None):
    self.okButton.enabled = (bool(self.nameText.text) and
                             bool(self.emailText.text))
```

- Questo altro metodo di Form viene invocato ogni volta che viene cliccato un bottone.
- In questo esempio il metodo si limita a stampare OK o Cancel ma nelle applicazioni reali ovviamente compie azioni piu` interessanti.

```
def clicked(self, widget):
    if widget == self.okButton:
        print("OK")
    elif widget == self.cancelButton:
        print("Cancel")
```


36

Il Design Pattern Mediator: un esempio

- Il metodo __init__() della classe Mediator
  - crea un dizionario di tipo defaultdict le cui chiavi sono widget e i cui valori sono liste di uno o piu` callable
  - Il for considera le coppie (widget, callable) presenti nella tupla passata come secondo argomento. Per ciascuno dei widget in queste coppie, viene inserito nel dizionario un elemento con chiave uguale al widget e valore uguale ad una lista vuota alla quale viene immediatamente aggiunto il caller associato al widget (si veda slide che illustra defaultdict).
  - Alla fine viene settato (ed eventualmente creato) l'attributo mediator del widget in modo che contenga il mediator appena creato.

```
class Mediator:

    def __init__(self, widgetCallablePairs):
        self.callablesForWidget = collections.defaultdict(list)
        for widget, caller in widgetCallablePairs:
            self.callablesForWidget[widget].append(caller)
            widget.mediator = self
```


Il Design Pattern Mediator: un esempio

- Ogni volta che un oggetto mediato (cioe` un widget passato a Mediator) cambia stato esso invoca il seguente metodo di Mediator che si occupa di invocare ogni metodo associato al widget.

```
    def on_change(self, widget):
        callables = self.callablesForWidget.get(widget)
        if callables is not None:
            for caller in callables:
                caller(widget)
        else:
            raise AttributeError("No on_change() method registered for {}"
                    .format(widget))
```

Il Design Pattern Mediator: un esempio

- Questa e` una classe usata come classe base per le classi mediate.
- Le istanze della classe mantengono un riferimento all'oggetto mediatore
- Il metodo Mediated.on_change() invoca il metodo on_change() del mediatore passandogli il widget mediato su cui e` stato invocato il metodo Mediated.on_change.
- Siccome questa classe non e` modificata dalle sue sottoclassi, essa rappresenta un esempio in cui e` possibile rimpiazzare la classe base con un decoratore di classe

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```


Il Design Pattern Mediator: un esempio

- La classe Button estende Mediated e di conseguenza un oggetto bottone ha l'attributo self.mediator e il metodo on_change che viene invocato quando il bottone cambia stato (ad esempio quando viene cliccato.
- In questo esempio, un'invocazione di Button.click() provoca un'invocazione di Button.on_change() (ereditato da Mediated), che a sua volta causa un'invocazione del metodo on_change() del mediatore.
  - Il metodo on_change() del mediatore invochera` i metodi associati al bottone. In questo caso, viene invocato il metodo Form.clicked() con il bottone stesso come argomento di tipo widget.

```
class Button(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.enabled = True
        self.text = text

    def click(self):
        if self.enabled:
            self.on_change()
```

Il Design Pattern Mediator: un esempio

- La classe Text ha la stessa struttura di Button.
- Per ogni widget (button widget, text widget, ecc.), il fatto di definire la classe corrispondente come sottoclasse di Mediated permette di lasciare al mediatore il compito di occuparsi delle azioni legate ad un cambio di stato del widget.
- Ovviamente quando si crea il mediatore occorre stabilire le associazioni tra i widget e i metodi che vogliamo vengano invocati

```
class Text(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.__text = text

    @property
    def text(self):
        return self.__text

    @text.setter
    def text(self, text):
        if self.text != text:
            self.__text = text
            self.on_change()
```


Il Design Pattern Mediator: un esempio

```
def main():
    form = Form()
    test_user_interaction_with(form)

def test_user_interaction_with(form):
    form.okButton.click()   # Ignorato perche' bottone disabiliato  dalla chiamata a self.update_ui() in create_mediator()
    print(form.okButton.enabled)   # False
    form.nameText.text = "Fred"
    print(form.okButton.enabled)   # False perche' non basta aver inserito solo il nome
    form.emailText.text = "fred@bloggers.com"
    print(form.okButton.enabled)   # True perche; è stato inserito anche l'indirizzo mail
    form.okButton.click()          # OK
    form.emailText.text = ""
    print(form.okButton.enabled)   # False
    form.cancelButton.click()      # Cancel

if __name__ == "__main__":
    main()
```

Un programma che crea e usa una form

Il Design Pattern Mediator: un esempio basato su coroutine

- Un mediatore si presta ad un'implementazione mediante coroutine perche' puo` essere visto come una pipeline che riceve messaggi (derivanti da invocazioni di on_change()) e passa questi messaggi agli oggetti interessati.
- In questo esempio viene implementato un mediator mediante coroutine per lo stesso problema considerato nell'esempio precedente.
- A differenza di quanto accadeva prima, in questa implementazione ogni widget e` associato ad un mediatore che e` una pipeline di coroutine (prima il mediatore era un oggetto associato all'intera form e tutti i widget della form erano associati insieme ai rispettivi callable al mediatore).
  - Ogni volta che un widget cambia stato (ad esempio, viene cliccato un bottone), esso invia se stesso alla pipeline.
  - Sono le componenti della pipeline a decidere se vogliono svolgere o meno azioni in risposta al cambio di stato del widget.
- Nell'approccio precedente il metodo on_change() del mediatore invoca i metodi associati al widget nel caso in cui il widget cambia stato
- Il codice non illustrato e` identico a quello visto nell'esempio precedente..

43

Il Design Pattern Mediator: un esempio basato su coroutine

- Non abbiamo bisogno di una classe Mediator in quanto il mediator e` di fatto una pipeline di coroutine
- Il metodo in basso crea una pipeline di coroutine di due componenti, self._update_ui_mediator() e self._clicked_mediator().
- Una volta creata la pipeline, l'attributo mediator della pipeline viene settato con questa pipeline.
- Alla fine, viene inviato None alla pipeline e siccome nessun widget e` None, nessuna azione specifica sara` intrapresa ad eccezione di azioni che interessano la form (come per esempio abilitare o disabilitare il bottone OK in _update_ui_mediator()).

```
    def create_mediator(self):
        self.mediator = self._update_ui_mediator(self._clicked_mediator())
        for widget in (self.nameText, self.emailText, self.okButton,
                       self.cancelButton):
            widget.mediator = self.mediator
        self.mediator.send(None)
```


44

Il Design Pattern Mediator: un esempio basato su coroutine

- Questa coroutine e` parte della pipeline.
- Ogni volta che un widget notifica un cambio di stato, il widget passato alla pipeline e ` restituito dall'espressione yield e salvato nella variabile widget.
- **Quando occorre abilitare o disabilitare il bottone ok, questo viene fatto indipendentemente da quale widget abbia cambiato stato.**
  - Potrebbe anche non essere cambiato lo stato di nessun widget, cioe` che il widget sia None e quindi che la form sia stata semplicemente inizializzata.
  - Dopo aver settato il campo enabled del bottone, la coroutine passa il widget alla chain

```
@coroutine
def _update_ui_mediator(self, successor=None):
    while True:
        widget = (yield)
        self.okButton.enabled = (bool(self.nameText.text) and
                                 bool(self.emailText.text))
        if successor is not None:
            successor.send(widget)
```


45

Il Design Pattern Mediator: un esempio basato su coroutine

- Questa coroutine si occupa solo dei click dei bottoni Ok e Cancel
- Se uno di questi bottoni e` il widget che ha cambiato stato allora questa coroutine gestisce il cambio di stato altrimenti passa il widget alla prossima coroutine nella pipeline, se ve ne e` una.

```
@coroutine
def _clicked_mediator(self, successor=None):
    while True:
        widget = (yield)
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
        elif successor is not None:
            successor.send(widget)
```


46

Il Design Pattern Mediator: un esempio basato su coroutine

- Le classi Text e Button sono le stesse dell'implementazione basata sull'approccio convenzionale.
- La classe Mediated e` lievemente diversa in quanto il suo metodo on_change() invia il widget che ha cambiato stato alla pipeline.

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.send(self)
```


Programmazione Avanzata

Design Pattern: Observer

IL Design Pattern Observer

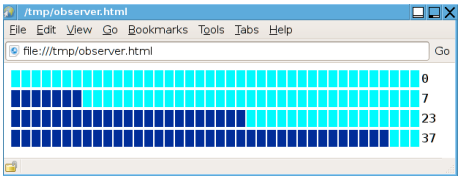
- Il pattern Observer e` un design pattern comportamentale che supporta relazioni di dipendenza tra oggetti in modo tale che quando un oggetto cambia stato tutti gli oggetti collegati sono informati del cambio.
- Tipicamente si tratta di gestire una relazione di dipendenza one-to-many tra un oggetto osservato e degli osservatori e l'obiettivo e` di fare in modo che gli oggetti non siano strettamente accoppiati.
  - In alcuni casi potrebbe trattarsi di una relazione many-to-many (piu` di un oggetto osservato)
- Il design pattern observer evita che l'oggetto osservato aggiorni direttamente gli oggetti ad esso collegati in quanto cio` determinerebbe un accoppiamento stretto tra l'oggetto osservato e sui osservatori.

IL Design Pattern Observer:un esempio

- Uno degli esempi di questo pattern e delle sue varianti e` il paradigma model/view/controller (MVC) che consiste nel separare un'applicazione in tre componenti logiche: modello, view e controller.
  - Il modello gestisce i dati e la logica dell'applicazione indipendentemente dall'interfaccia utente
  - una o piu` view visualizzano i dati in una o piu` forme comprensibili per l'utente
    - Ogni cambio nel modello si riflette automaticamente nelle view associate
  - uno o piu` controller mediano tra input e modello, cioe` converte l'input in comandi per il modello o le view.
- Una popolare semplificazione dell'approccio MVC consiste nell'usare un paradigma model/view dove le view si occupano sia di visualizzare i dati sia di mediare tra input e modello.
  - In termini di Observer Pattern cio` significa che le view sono osservatori del modello e il modello e` l'oggetto dell'osservazione.

IL Design Pattern Observer: un esempio

- Consideriamo un modello che rappresenta un valore con un minimo e un massimo, come ad esempio una scrollbar o un controllo della temperatura.
- Vengono creati due osservatori (view) separati per il modello: uno per dare in output il valore del modello ogni volta che esso cambia sotto forma di una barra di progressione in formato HTML, l'altro per mantenere la storia dei cambiamenti (valori e timestamp).




```
$ ./observer.py > /tmp/observer.html
  0 2013-04-09 14:12:01.043437
  7 2013-04-09 14:12:01.043527
 23 2013-04-09 14:12:01.043587
 37 2013-04-09 14:12:01.043647
```


51

IL Design Pattern Observer: un esempio

- La classe Observed e` estesa dai modelli o da ogni altra classe che supporta l'osservazione.
- La classe Observed mantiene un insieme di oggetti osservatori.
  - Ogni volta che viene aggiunto un oggetto osservatore all'oggetto osservato, il metodo update() dell'osservatore e` invocato per inizializzare l'osservatore con lo stato attuale del modello.
  - Se in seguito il modello cambia stato, esso invoca il metodo observers_notify() in modo tale che il metodo update() di ogni osservatore possa essere invocato per assicurare che ogni osservatore (view) rappresenti il nuovo stato del modello.

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)

    def observer_discard(self, observer):
        self.__observers.discard(observer)

    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)
```

52

IL Design Pattern Observer: un esempio

- Il metodo observers_add()
  - accetta uno o piu` osservatori da aggiungere. Per questo motivo oltre a *observer c'e` il parametro observer che assicura che il numero di osservatori passati in input al metodo non sia zero.
  - usa nel for il metodo **intertools.chain(*iterables)** che crea un iteratore che restituituisce gli elementi dall'oggetto iterabile specificato come primo argomento e quando non ci sono piu` elementi da restituire in questa lista, passa alla prossima collezione iterabile e cosi` via fino a che non vengono restituiti gli elementi di tutte le collezioni iterabili in iterables. Il for avrebbe potuto usare la concatenazione di tuple in questo modo "for observer in (observer,) + observers:"

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)

    def observer_discard(self, observer):
        self.__observers.discard(observer)

    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)
```


IL Design Pattern Observer: un esempio

- La classe SliderModel eredita dalla classe Observed un insieme privato di osservatori che inizialmente e` vuoto e i metodi observers_add(), observer_discard() e observers_notify()
- Quando lo stato del modello cambia, per esempio quando il suo valore cambia, esso deve invocare il metodo observers_notify() in modo che ciascun osservatore possa rispondere di conseguenza.
- SliderModel ha anche le proprieta` minimum e maximum i cui setter, come quello di value, invocano il metodo observers_notify().

```
class SliderModel(Observed):

    def __init__(self, minimum, value, maximum):
        super().__init__()
        # These must exist before using their property setters
        self.__minimum = self.__value = self.__maximum = None
        self.minimum = minimum
        self.value = value
        self.maximum = maximum

    @property
    def value(self):
        return self.__value

    @value.setter
    def value(self, value):
        if self.__value != value:
            self.__value = value
            self.observers_notify()
    ...
```

IL Design Pattern Observer: un esempio

- HistoryView e` un osservatore del modello e per questo fornisce un metodo update() che accetta il modello osservato come suo unico argomento (oltre self).
- Ogniqualvolta il metodo update() e` invocato, esso aggiunge una tupla (value, timestamp) alla sua self.data list, mantenendo in questo modo la storia di tutti i cambiamenti applicati al modello.

```
class HistoryView:

    def __init__(self):
        self.data = []

    def update(self, model):
        self.data.append((model.value, time.time()))
```


IL Design Pattern Observer: un esempio

- Questa e` un'altra view per osservare il modello. L'attributo length rappresenta il numero di celle usate per rappresentare il valore del modello in una riga della tabella HTML.
- Il metodo update viene invocato quando il modello e` osservato per la prima volta e quando viene successivamente aggiornato
  - Il metodo stampa una tabella HTML di una riga con un numero self.length di celle per rappresentare il modello. Le celle sono di colore ciano se sono vuote e blu scuro altrimenti.

```
class LiveView:

    def __init__(self, length=40):
        self.length = length
```

```
    def update(self, model):
        tippingPoint = round(model.value * self.length /
                (model.maximum - model.minimum))
        td = '<td style="background-color: {}"> </td>'
        html = ['<table style="font-family: monospace" border="0"><tr>']
        html.extend(td.format("darkblue") * tippingPoint)
        html.extend(td.format("cyan") * (self.length - tippingPoint))
        html.append("<td>{}</td></tr></table>".format(model.value))
        print("".join(html))
```

#si puo` sostituire la print con:
o=open("output.html","a")
o.write("".join(html))
o.close()
webbrowser.open("file:///Users/.../output.html")