Programmazione Avanzata

Design Pattern (V parte)

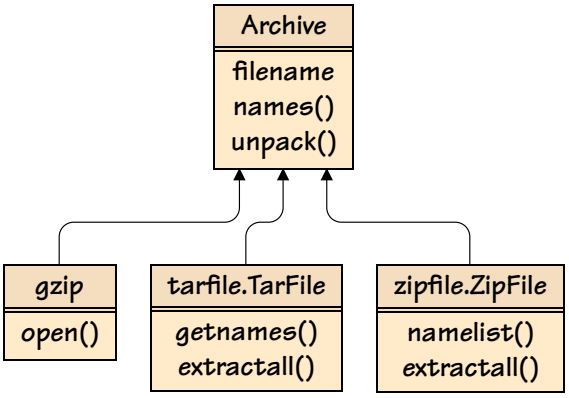
1

Il Design Pattern Facade

- Il design pattern Facade è un design pattern strutturale che fornisce un'interfaccia semplificata per un sistema costituito da interfacce o classi troppo complesse o troppo di basso livello.
- Esempio: La libreria standard di Python fornisce moduli per gestire file compressi gzip, tarballs e zip. Questi moduli hanno interfacce diverse.
- Immaginiamo di voler accedere ai nomi di un file di archivio ed estrarre i suoi file usando un'interfaccia semplice.
- Soluzione: Usiamo il design pattern Facade per fornire un'interfaccia semplice e uniforme che delega la maggior parte del vero lavoro alla libreria standard.

2

Il Design Pattern Facade: un esempio



3

Il Design Pattern Facade: un esempio

```
class Archive:

    def __init__(self, filename):
        self._names = None
        self._unpack = None
        self._file = None
        self.filename = filename
```

- La variabile self._names è usata per contenere un callable che restituisce una lista dei nomi dell'archivio.
- La variabile the self._unpack è usata per mantere un callable che estrae tutti i flie dell'archivio nella directory corrente.
- La variabile self._file è usata per mantenere il file object che è stato aperto per l'archivio.
- self.filename è una proprietà che mantiene il nome del file di archivio.

4

Il Design Pattern Facade: un esempio

```
    @property
    def filename(self):
        return self.__filename

    @filename.setter
    def filename(self, name):
        self.close()
        self.__filename = name
```

Se l'utente cambia il nome del file, ad esempio archive.filename = newname, allora il file d'archivio corrente, se aperto, viene chiuso e viene aggiornata la variabile __filename.
Non viene immediatamente aperto il nuovo archivio, in quanto la classe Archive apre l'archivio solo se necessario.

Il Design Pattern Facade: un esempio

```
    def close(self):
        if self._file is not None:
            self._file.close()
        self._names = self._unpack = self._file = None
```

Gli utenti della classe Archive invocano close() quando hanno finito con un'istanza.
Il metodo chiude il file object , se c'è un file object aperto, e setta self._names, self._unpack, e self._file a None per invalidarli.

La classe Archive è un context manager (tra un po' maggiori dettagli) e così in pratica gli utenti non hanno bisogno di chiamare close(), a patto che usino la classe in uno statement with, come nel codice qui in basso:

```
with Archive(zipFilename) as archive:
    print(archive.names())
    archive.unpack()
```

Il Design Pattern Facade: un esempio

```
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.close()
```

- Questi due metodi rendono un Archivio un context manager
- Il metodo __enter__() method restituisce self (un'istanza di Archive) che viene assegnata alla variabile dello statement with ...as
- Il metodo __exit__() chiude il file object dell'archivio se c'è ne uno aperto..

7

Il Design Pattern Facade: un esempio

```
    def names(self):
        if self._file is None:
            self._prepare()
        return self._names()
```

Questo metodo restituisce una lista dei nomi dei file dell'archivio aprendo l'archivio (se non è già aperto) e ponendo in self._names e self._unpack dei callable appropriati utilizzando il metodo self.prepare().

8

Il Design Pattern Facade: un esempio

```
    def unpack(self):
        if self._file is None:
            self._prepare()
        self._unpack()
```

Questo metodo spacchetta tutti i file di archivio ma solo se tutti i loro nomi sono "safe".

9

Il Design Pattern Facade: un esempio

```
    def _prepare(self):
        if self.filename.endswith((".tar.gz", ".tar.bz2", ".tar.xz",
                ".zip")):
            self._prepare_tarball_or_zip()
        elif self.filename.endswith(".gz"):
            self._prepare_gzip()
        else:
            raise ValueError("unreadable: {}".format(self.filename))
```

Questo metodo delega la preparazione ai metodi adatti a occuparsene,
Per i tarball e i file zip il codice necessario è molto simile e per questo essi vengono preparati dallo stesso metodo.
I file gzip richiedono una gestione diversa e per questo hanno un metodo a parte.
I metodi di preparazione devono assegnare dei callable alle variabili self._names e self._unpack in modo che queste possano essere chiamate nei metodi names() e unpack().

10

Il Design Pattern Facade: un esempio

- Questo metodo comincia con il creare una funzione innestata safe_extractall() che controlla tutti i nomi dell'archivio e lancia ValueError se qualcuno di essi non è safe.
- Se tutti i nomi sono safe viene invocato o il metodo tarball.TarFile.extractall() oppure il metodo zipfile.ZipFile.extractall().

```
    def _prepare_tarball_or_zip(self):
        def safe_extractall():
            unsafe = []
            for name in self.names():
                if not self.is_safe(name):
                    unsafe.append(name)
            if unsafe:
                raise ValueError("unsafe to unpack: {}".format(unsafe))
            self._file.extractall()
        if self.filename.endswith(".zip"):
            self._file = zipfile.ZipFile(self.filename)
            self._names = self._file.namelist
            self._unpack = safe_extractall
        else: # Ends with .tar.gz, .tar.bz2, or .tar.xz
            suffix = os.path.splitext(self.filename)[1]
            self._file = tarfile.open(self.filename, "r:" + suffix[1:])
            self._names = self._file.getnames
            self._unpack = safe_extractall
```


Il Design Pattern Facade: un esempio

- A seconda dell'estensione del nome dell'archivio, viene aperto un tarball.TarFile o uno zipfile.ZipFile e assegnato a self._file.
- self._names viene settata al metodo bound corrispondente (namelist() o getnames())
- self._unpack viene settata alla funzione safe_extractall() appena creata. Questa funzione è una chiusura che ha catturato self e quindi può accedere a self._file e chiamare il metodo appropriato extractall()

```
    def _prepare_tarball_or_zip(self):
        def safe_extractall():
            unsafe = []
            for name in self.names():
                if not self.is_safe(name):
                    unsafe.append(name)
            if unsafe:
                raise ValueError("unsafe to unpack: {}".format(unsafe))
            self._file.extractall()
        if self.filename.endswith(".zip"):
            self._file = zipfile.ZipFile(self.filename)
            self._names = self._file.namelist
            self._unpack = safe_extractall
        else: # Ends with .tar.gz, .tar.bz2, or .tar.xz
            suffix = os.path.splitext(self.filename)[1]
            self._file = tarfile.open(self.filename, "r:" + suffix[1:])
            self._names = self._file.getnames
            self._unpack = safe_extractall
```

Il Design Pattern Facade: un esempio

```
    def is_safe(self, filename):
        return not (filename.startswith(("/", "\\")) or
            (len(filename) > 1 and filename[1] == ":" and
             filename[0] in string.ascii_letter) or
            re.search(r"[.][.][/\\]", filename))
```

- Un file di archivio creato in modo malizioso potrebbe, una volta spacchettato, sovrascrivere importanti file di sistema rimpiazzandoli con file non funzionanti o pericolosi.
- In considerazione di ciò, non dovrebbero mai essere aperti archivi contenenti file con path assoluti o che includono path relative ed evitare di aprire gli archivi con i privilegi di un utente come root o Administrator.
- is_safe() restituisce False se il nome del file comincia con un forward slash o con un backslash (cioè un path assoluto) o contiene ../ o ..\ (cioè un path relativo che potrebbe condurre ovunque), oppure comincia con D: dove D indica un'unità disco di Windows.

13

Il Design Pattern Facade: un esempio

```
    def _prepare_gzip(self):
        self._file = gzip.open(self.filename)
        filename = self.filename[:-3]
        self._names = lambda: [filename]
        def extractall():
            with open(filename, "wb") as file:
                file.write(self._file.read())
        self._unpack = extractall
```

Questo metodo fornisce un object file aperto per self._file e assegna callable adatti a self._names e self._unpack.
La funzione extractall(), legge e scrive dati.
Il pattern Facade permette di creare interfacce semplici e comode che ci permettono di ignorare i dettagli di basso livello. Uno svantaggio di questo design pattern potrebbe essere quello di non consentire un controllo più fine.
Tutttavia, un facade non nasconte o elimina le funzionalità del sistema sottostante e così è possibile usare un facade passando però a classi di più basso livello se abbiamo bisogno di un maggiore controllo.

14

I context manager

I context manager consentono di allocare e rilasciare risorse quando vogliamo
L'esempio più usato di context manager è lo statement with.

```
with open('some_filé, 'w') as opened_file:
        opened_file.write('Hola!')
```

Questo codice apre il file, scrive alcuni dati in esso e lo chiude. Se si verifica un errore mentre si scrivono i dati, esso cerca di chiuderlo.
Il codice in alto è equivalente a

```
file = open('some_filé, 'w')
try:
        file.write('Hola!')
finally:
        file.close()
```


15

I context manager

- è possibile implementare un context manager con una classe.

```
class File:
        def __init__(self, file_name, method):
                self.file_obj = open(file_name, method)
        def __enter__(self):
                return self.file_obj
        def __exit__(self, type, value, traceback):
                self.file_obj.close()
```


16

I context manager

- è sufficiente definire __enter__() ed __exit__() per poter usare la classe File in uno statement with.

```
with File('demo.txt', 'w') as opened_file:
        opened_file.write('Hola!')
```


17

I context manager

- Come funziona lo statement with:
  - Immagazzina il metodo __exit__() della classe File
  - Invoca il metodo __enter__() della classe File
    - Il metodo __enter__ restituisce il file object per il file aperto.
  - L'object file è passato a opened_file.
  - Dopo che è stato eseguito il blocco al suo interno, lo statement with invoca il metodo __exit__()
    - Il metodo __exit__() chiude il file

```
with File('demo.txt', 'w') as opened_file:
        opened_file.write('Hola!')
```


18

I context manager

- Se tra il momento in cui viene passato l'object file a opened_file e il momento in cui viene invocata __exit__, si verifica un'eccezione allora Python passa type, value e traceback dell'eccezione come argomenti a __exit__() per decidere come chiudere il file e se eseguire altri passi. In questo esempio gli aromenti di exit non influiscono sul suo comportamento.

```
with File('demo.txt', 'w') as opened_file:
        opened_file.write('Hola!')
```


19

I context manager

- Se il file object lanciasse un'eccezione, come nel caso in cui provassimo ad accedere ad un metodo non supportato dal file object:

```
with File('demo.txt', 'w') as opened_file:
        opened_file.undefined_function('Hola!')
```

- with eseguirebbe i seguenti passi:

1. passerebbe type, value e traceback a __exit__()
2. permetterebbe a __exit__() di gestire l'eccezione
3. Se __exit__() restituisse True allora l'eccezione non verrebbe rilanciata dallo statement with.
4. Se __exit__() restituisse un valore diverso da True allora l'eccezione verrebbe lanciata dallo statement with

20

I context manager

```
with File('demo.txt', 'w') as opened_file:
        opened_file.undefined_function('Hola!')
```

Nel nostro esempio, __exit__() restituisce (implicitamente) None per cui with lancerebbe l'eccezione:

```
Traceback (most recent call last):
      File "<stdin>", line 2, in <module>
AttributeError: 'filé object has no attribute 'undefined_function'
```


21

I context manager

Il metodo __exit__() in basso invece gestisce l'eccezione:

```
class File(object):
        def __init__(self, file_name, method):
                self.file_obj = open(file_name, method)
        def __enter__(self):
                return self.file_obj
        def __exit__(self, type, value, traceback):
                print("Exception has been handled")
                self.file_obj.close()
                return True

with File('demo.txt', 'w') as opened_file:
        opened_file.undefined_function()
```


22

I context manager

- è possibile implementare un context manager con un generatore utilizzando il modulo contextlib. Il decoratore Python contextmanager trasforma il generatore open_file in un oggetto GeneratorContextManager

```
from contextlib import contextmanager
@contextmanager
def open_file(name):
        f = open(name, 'w')
        yield f
        f.close()
```

- Si usa in questo modo

```
with open_file('some_filé) as f:
        f.write('hola!')
```


I context manager

- Nel punto in cui c'è yield il blocco nello statement with viene eseguito.
- Il generatore riprende all'uscita del blocco.
- Se nel blocco si verifica un'eccezione non gestita, essa viene rilanciata nel generatore nel punto dove si trova yield.
- è possibile usare uno statement try...except...finally per catturare l'errore.
- Se un'eccezione è catturata solo affinché al fine di registrarla o per svolgere qualche azione (piuttosto che per sopprimerla), il generatore deve rilanciare l'eccezione. Altrimenti il generatore context manager indicherà allo statement with che l'eccezione è stata gestita e l'esecuzione riprenderà dallo statement che segue lo statement with.

I context manager

- Lo statement try...finally garantisce che il file venga chiuso anche nel caso si verifichi un' eccezione nel blocco del with

```
from contextlib import contextmanager
@contextmanager
 def open_file(name):
        f = open(name, 'w')
        try:
                yield f
        finally:
                f.close()
```


25

Il Design Pattern Flyweight

- Il pattern Flyweight è concepito per gestire un grande numero di oggetti relativamente piccoli dove molti degli oggetti sono duplicati l'uno dell'altro.
- Il pattern è implementato in modo da avere un'unica istanza per rappresentare tutti gli oggetti uguali tra loro. Ogni volta che è necessario, questa unica istanza viene condivisa.
- Python permette di implementare Flyweight in modo naturale grazie all'uso dei riferimenti. Ad esempio, una lunga lista di stringhe molte delle quali sono duplicati potrebbe richiedere molto meno spazio se al posto delle stringhe venissero memorizzati i riferimenti ad esse.

```
red, green, blue = "red", "green", "blue"
x = (red, green, blue, red, green, blue, red, green)
y = ("red", "green", "blue", "red", "green", "blue", "red", "green")
```

Nel codice in alto, x immagazzina 3 stringhe usando 8 riferimenti mentre la tupla y immagazzina 8 stringhe usando 8 riferimenti dal momento che quello che abbiamo scritto corrisponde a scrivere _anonymous_item0 = "red", …, _anonymous_item7 = "green"; y = (_anonymous_item0, …_anonymous_item7).

26

Il Design Pattern Flyweight

- Probabilmente il modo più semplice per trarre vantaggio dal patter Flyweight in Python è di usare un dict, in cui ciascun oggetto (unico) corrisponde ad un valore identificato da un'unica chiave.
- Ciò assicura che ciascun oggetto distinto viene creato un'unica volta, indipendentemente da quante volte viene usato.
- In alcune situazioni si potrebbero avere molti oggetti non necessariamente piccoli dove gran parte di essi o tutti sono unici. Un facile modo per ridurre l'uso della memoria in questo è di usare __slots__

__slots__

- In Python ogni classe può avere attributi di istanza.
- Per default Python usa un dict per immagazzinare gli attributi di instanza di un oggetto. Ciò è molto utile perché consente di settare nuovi attributi durante l'esecuzione.
- Comunque per classi piccole con attributi noti questo comportamento potrebbe essere un collo di bottiglia in quanto il dict comporterebbe uno spreco di RAM nel caso in cui vengano creati molti oggetti.

__slots__

- Un modo per evitare questo spreco di RAM e di usare __slots__ per indicare a Python di non usate un dict, e di allocare spazio solo per un insieme fissato di attributi.

- __slots__ è una variabile di classe a cui può essere assegnata una stringa, un iterabile, o una sequenza di stringhe.
- *__slots__* riserva spazio per le variabili dichiarate e previene la creazione automatica di *__dict__* (e di *__weakref__)* per ciascuna istanza.

__slots__

- Vediamo un esempio di implementazione della stessa classe con e senza __slots__ .
- Senza __slots__

```
class MyClass(object):
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
```

__slots__

• con __slots__

```
class MyClass(object):
    __slots__ = ['namé, 'identifier’]
    def __init__(self, name, identifier):
        self.name = name
        self.identifier=identifier
```

Non posso aggiungere altre variabili di istanza alle istanza di MyClass

31

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ("x", "y", "z", "color")

    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

- La classe Point mantiene una posizione nello spazio tridimensionale e un colore.
- Grazie a __slots__, nessun Point ha il suo dict (self.__dict__) privato.
- Ciò vuol dire che nessun attributo può essere aggiunto a punti individuali.
- Un programma per creare una tupla di un milione di punti ha impiegato su una stessa macchina
- nella versione con slots, circa 2 secondi e il programma ha occupato 183 Mebibyte di RAM
- nella versione senza slots, una frazione di secondi in meno ma il programma ha occupato 312 Mebibyte di RAM.

- Per default Python sacrifica sempre la memoria a favore della velocità ma è sempre possibile invertire queste priorità se è conveniente farlo.

32

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

Questo è l'inizio di un'altra classe Point.
Essa utilizza un database DBM (chiave-valore) immagazzinato in un file su disco.
Un riferimento al DBM è mantenuto nella variabile Point.__dbm.
Tutti i punti condividono lo stesso file DBM.

Uno "shelf" è un oggetto persistente simile ad un dizionario. I valori (non le chiavi) in uno shelf possono essere arbitrari oggetti gestibili dal modulo pickle. Ciò include la maggior parte di istanze di classi, tipi di dati ricorsivi, e oggetti contenenti molti oggetti condivisi.
Le chiavi sono stringhe.
shelve.open(*filename*, *flag='c'*, *protocol=None*, *writeback=False*) apre un dizionario persistente.
Il filename specificato è il nome di base per il database sottostante.
Per default il file database è aperto in lettura e scrittura.

33

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

tempfile.gettempdir() restituisce il nome della directory usata per i file temporanei.

Il comportamento di default di open fa in modo che venga creato il file DBM se non esiste già .
Il modulo shelve serializza i valori immagazzinati e li deserializza quando i valori vengono recuperati dal database.
Il processo di deserializzazione in Python non è sicuro perché esegue dell'arbitrario codice Python e di conseguenza non dovrebbe mai essere effettuato su dati provenienti da fonti non affidabili,

34

Il Design Pattern Flyweight

```
    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

A differenza del metodo __init__() della prima classe Point, questo metodo assegna i valori delle variabili in un file DBM.

```
    def __getattr__(self, name):
        return Point.__dbm[self.__key(name)]
```

Questo metodo è invocato ogni volta che si accede ad un attributo della classe

35

Il Design Pattern Flyweight

```
def __key(self, name):
    return  "{:X}:{}".format(id(self), name)
```

Questo metodo fornisce una stringa chiave per ognuno degli attributi x, y, z e color.
La chiave è ottenuta dall'ID restituita da id(self) in esadecimale e dal nome dell'attributo. Per esempio se l'ID di un punto è 3954827, il suo attributo x avrà chiave "3C588B:x", il suo attributo y avrà chiave "3C588B:y", e così via.

36

Il design pattern Flyweight

- Le chiavi e i valori dei database DBM devono essere byte.
- Per fortuna, i moduli DBM Python accettano sia str che byte come chiavi convertendo le stringhe in byte.
- In particolare, il modulo shelve, qui usato, permette di immagazzinare un qualsiasi valore gestibile dal modulo pickle.
- Un valore recuperato dal database è convertito dalla rappresentazione sotto forma di sequenza di bytes nel tipo originario.

37

Il Design Pattern Flyweight

```
    def __setattr__(self, name, value):
        Point.__dbm[self.__key(name)] = value
```

Ogni volta che un attributo di Point è settato (ad esempio, point.y = y), viene invocato questo metodo.
Il valore value immagazzinato è convertito in un flusso di byte.

38

Il Design Pattern Flyweight

Sulla macchina usata per i test, la creazione di un milione di punti ha richiesto circa un minuto ma il programma ha occupato solo 29 Mebibyte of RAM (più 361 Mebibyte di spazio su disco) mentre la prima versione di Point ha richiesto 183 Mebibyte di RAM.

Il pattern Prototype

- Il pattern Prototype è un design pattern creazionale usato per creare nuovi oggetti clonando un oggetto preesistente e poi modificando il clone così creato.

Il pattern Prototype: esempio

- Point è la classe preesistente

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```


41

Il pattern Prototype: esempio

- In questo codice cloniamo nuovi punti in diversi modi

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14) # Could have used any of point1 to point6
```


42

Il pattern Prototype: esempio

- point1 = Point(1, 2)
  - viene semplicemente invocato il costruttore della classe Point.
  - point 1 è creato in modo statico. Di seguito creeremo istanze di Point in modo dinamico.
- point2 = **eval**("{}({}, {})".format("Point", 2, 4))
  - usa eval() per creare istanze di Point
  - eval valuta l'espressione Python rappresentata dalla stringa ricevuta come argomento. In altre parole, esegue il codice passato come argomento

Il pattern Prototype: esempio

- point3 = **getattr**(sys.modules[__name__], "Point")(3, 6)
  - usa getattr() per creare un'istanza
  - **getattr(object,name, default)** restituisce il valore dell'attributo dell'oggetto
    - **object** : oggetto per il quale viene restituito il valore dell'attributo nominato
    - **name** : stringa che contiene il nome dell'attributo
    - **default (opzionale):** valore restituito quando l'attributo specificato non viene trovato
  - nel codice in alto
  - **sys.modules** è un dizionario che mappa i nomi dei moduli ai moduli che sono già stati caricati
  - l'espressione sys.modules[__name__] restituisce il modulo in cui essa si trova
  - l'espressione **getattr**(sys.modules[__name__], "Point") restituisce il valore dell'attributo Point del modulo

Il pattern Prototype: esempio

- point4 = **globals**()["Point"](4, 8)

  - La funzione built-in globals() restituisce un dizionario che rappresenta la tavola dei simboli globali. All'interno di una funzione o un metodo, il dizionario è quello relativo al modulo dove è definita la funzione (o il metodo), non quello in cui è invocata .
  - comportamento similie a **getattr(object,name, default)**

Il pattern Prototype: esempio

- point5 = **make_object**(Point, 3,9)

  - usa la funzione make-object

Il pattern Prototype: esempio

- point6 = copy.deepcopy(point5)
  - usa il classico approccio basato su Prototype:
    - prima clona un oggetto esitente
    - poi lo inizializza con le istruzioni successive

- point7 = point1.__class__(7, 14)
  - point7 è creato usando point1
  - instanza.__class__ contiene la classe a cui appartiene istanza

Python ha un support built-in per creare oggetti sulla base di prototipi: la funzione copy.deepcopy().

47

Il Design Pattern State: un esempio

- Il main() comincia con il creare dei contatori. **Le istanze così create sono callable** e quindi possono essere usate come funzioni.
  - Le istanze di Counter mantengono contatori separati per ciascuno dei nomi passati come argomento o, in assenza di un nome (come totalCounter), matengono un contatore singolo.
- Viene quindi creato un multiplexer (che per default è attivo) e vengono connesse le funzioni callback agli eventi.
- I nomi degli eventi considerati sono “cars”, “vans” e “trucks”.
  - Nel for, la funzione carCounter() è connessa all’evento “cars”, la funzione commercialCounter() è connessa agli eventi “vans” e “trucks” e totalCounter() è connessa a tutti e tre gli eventi.

```
totalCounter = Counter()
carCounter = Counter("cars")
commercialCounter = Counter("vans", "trucks")

multiplexer = Multiplexer()
for eventName, callback in (("cars", carCounter),
        ("vans", commercialCounter), ("trucks", commercialCounter)):
    multiplexer.connect(eventName, callback)
    multiplexer.connect(eventName, totalCounter)
```

48

Il Design Pattern State: un esempio

- Con il codice mostrato in basso, main() genera 100 eventi random e li invia al multiplexer.
- Per un evento “cars”, il multiplexer invoca carCounter() e totalCounter(), passando l’evento come unico argomento a ciascuna chiamata. Se l’evento è invece “vans” o “trucks”, il multiplexer invoca le funzioni commercialCounter() e totalCounter().

```
for event in generate_random_events(100):
    multiplexer.send(event)
print("After 100 active events:  cars={} vans={} trucks={} total={}"
        .format(carCounter.cars, commercialCounter.vans,
                commercialCounter.trucks, totalCounter.count))
```