La guida completa alla Zope Component Architecture

Release 0.5.8

Baiju M e la comunità Plone Italia

April 22, 2016

Contents

Autore Baiju M

Versione 0.5.8

Libro stampato http://www.lulu.com/content/1561045

Online PDF (en) http://www.muthukadan.net/docs/zca.pdf

Traduttore Giacomo Spettoli <giacomo.spettoli@gmail.com>

Ringraziamenti

Molte persone mi hanno aiutato nella stesura di questo libro. La bozza iniziale fu revisionata dal mio collega Brad Allen. Quando annunciai questo libro attraverso il mio blog, ricevetti molti commenti di incoraggiamento a procedere con questo lavoro. Kent Tenney modificò numerose parti di questo libro e riscrisse anche l'applicazione di esempio. Molti altri mi hanno inviato correzioni e commenti, inclusi Lorenzo Gil Sanchez, Michael Haubenwallner, Nando Quintana, Stephane Klein, Tim Cook, Kamal Gill and Thomas Herve. Lorenzo ha tradotto questo lavoro in Spagnolo e Stephane in Francese. Grazie a tutti!

CHAPTER 1

Come iniziare

1.1 Introduzione

Sviluppare un sistema software di grandi dimensioni è sempre molto complicato. È stato dimostrato che quando si tratta di grandi sistemi un buon approccio all'analisi, al design e alla programmazione è dato dalla programmazione orientata agli oggetti. Il design basato sui componenti e la programmazione a componenti stanno diventando molto popolari in questi giorni. L'approccio basato sui componenti aiuta a scrivere e a mantenere facilmente testabili con unit-test i sistemi software. Ci sono molti framework per il supporto al design a componenti in diversi linguaggi, alcuni sono persino indipendenti dal linguaggio. Due esempi sono il COM della Microsoft e XPCOM di Mozilla.

La **Zope Component Architecture (ZCA)** è un framework Python per il supporto al design e alla programmazione basati sui componenti. Essa è molto adatta allo sviluppo di sistemi software di grandi dimensioni scritti in Python. La ZCA non è specifica per il web application server Zope: può essere utilizzata per qualsiasi applicazione Python. Forse dovrebbe essere chiamata *Python Component Architecture*.

La ZCA tratta principalmente l'utilizzo efficace degli oggetti Python. I componenti sono oggetti riutilizzabili con interfacce introspezionabili. Un interfaccia è un oggetto che descrive come interagire con un particolare componente. In altre parole: un componente fornisce un interfaccia implementata in una classe (o in qualsiasi altro oggetto chiamabile). Non è tanto importante come un oggetto venga implementato, l'importante è che esso aderisca al contratto della sua interfaccia. Utilizzando la ZCA, è possibile suddividere la complessità di un sistema su molteplici componenti che cooperano tra loro. Essa aiuta a creare due principali tipi di componenti: gli *adapter* e le *utility*.

I tre pacchetti principali che compongono la ZCA sono:

- `zope.interface`: viene utilizzato per definire l'interfaccia di un componente
- `zope.event`: fornisce un semplice sistema di eventi
- `zope.component`: si occupa della creazione, della registratione e del recupero dei componenti.

Notare che la ZCA non è un insieme di componenti, ma più propriamente serve a creare, registrare e recuperare i componenti. È bene ricordare inoltre che un *adapter* è una normale classe Python (o più in generale una factory) e una *utility* è un normale oggetto chiamabile Python.

Il framework ZCA fu sviluppato come parte del progetto Zope3. Come già anticipato, è un framework scritto esclusivamente in Python, così da poter essere utilizzato da qualsiasi tipo di applicazione Python. Attualmente i progetti Zope3, Zope2 e Grok utilizzano questo framework in maniera massiccia. Ci sono molti altri progetti che la utilizzano, inclusi progetti non legati al web [1].

[1] http://wiki.zope.org/zope3/ComponentArchitecture

1.2 Breve storia

Il progetto del framework ZCA iniziò nel 2011 come parte del progetto Zope3. Venne sviluppato basandosi sulle lezioni imparate durante lo sviluppo di grandi sistemi software utilizzando Zope2. Jim Fulton fu il project leader di questo progetto. Molte persone contribuirono al design e all'implementazione, inclusi ma non limitati a, Stephan Richter, Philipp von Weitershausen, Guido van Rossum (aka. Python BDFL), Tres Seaver, Phillip J Eby and Martijn Faassen.

Inizialmente la ZCA definiva dei componenti aggiuntivi, *services* e *views*, ma gli sviluppatori arrivarono alla conclusione che le utility potevano rimpiazzare i *service* e i multi-adapter potevano rimpiazzare le *view*. Oggi la ZCA ha un numero molto ridotto di tipi di componenti principali: *utility*, *adapter*, *subscriber*, e *handler*. In effetti, i *subscriber* e gli *handler* sono due particolari tipi di *adapter*.

Durante il ciclo di sviluppo di Zope3.2, Jim Fulton propose una grande semplificazione della ZCA [2]. Con questa semplificazione, fu creata una nuova singola interfaccia (*IComponentRegistry*) per la registrazione di componenti sia locali sia globali.

Il pacchetto `zope.component` ha una lunga lista di dipendenze, molte delle quali non erano necessarie per applicazioni non basate su Zope3. Durante il PyCon2007, Jim Fulton aggiunse a setuptools la funzionalità *extras_require* per permettere di separare il nucleo della ZCA dalle funzionalità aggiuntive [3].

Nel marzo del 2009, Tres Seaver rimosse poi le dipendenze da `zope.deferredimport` e `zope.proxy`.

Oggi, il progetto ZCA è un progetto indipendente con il proprio ciclo di rilasci e il proprio repository Subversion. Questo progetto sta diventando parte del più grande progetto del framework Zope [4]. In ogni caso le segnalazioni e i bug sono ancora tracciati come parte del progetto Zope3 [5], e la mailing list principale zope-dev viene utilizzata per le discussioni sullo sviluppo [6]. C'è anche un'altra user-list generica per Zope3 (zope3-users) che può essere utilizzata per qualsiasi domanda sulla ZCA [7].

1.3 Installazione

Il pacchetto `zope.component` insieme ai pacchetti `zope.interface` e `zope.event` costituiscono il nucleo della Zope Component architecture. Essi forniscono le strutture per definire, registrare e recuperare i componenti. Il pacchetto `zope.component` e le sue dipendenze sono disponibili in formato egg sul Python Package Index (PyPI) [8].

È possibile installare `zope.component` e le sue dipendenze utilizzando *easy_install* [9]

```
$ easy_install zope.component
```

Questo comando scarica `zope.component` e le sue dipendenze da PyPI e installa il tutto nel vostro *Python path*.

In alternativa, è possibile scaricare `zope.component` e le sue dipendenze da PyPI e poi installarle. Installare i pacchetti nell'ordine indicato sotto. Su sistemi Windows, potrebbero essere necessari i paccheti binari di `zope.interface`:

1. `zope.interface`
2. `zope.event`

[2] http://wiki.zope.org/zope3/LocalComponentManagementSimplification
[3] http://peak.telecommunity.com/DevCenter/setuptools#declaring-dependencies
[4] http://docs.zope.org/zopeframework/
[5] https://bugs.launchpad.net/zope3
[6] http://mail.zope.org/mailman/listinfo/zope-dev
[7] http://mail.zope.org/mailman/listinfo/zope3-users
[8] Repository dei pacchetti Python: http://pypi.python.org/pypi
[9] http://peak.telecommunity.com/DevCenter/EasyInstall

3. `zope.component`

Per installare questi pacchetti, dopo averli scaricati, è possibile usare il comando *easy_install* con gli eggs come argomento. (è possibile passare tutti gli egg come argomenti sulla stessa linea):

```
$ easy_install /path/to/zope.interface-3.x.x.tar.gz
$ easy_install /path/to/zope.event-3.x.x.tar.gz
$ easy_install /path/to/zope.component-3.x.x.tar.gz
```

È anche possibile installare questi pacchetti dopo averli estratti singolarmente. Ad esempio:

```
$ tar zxvf /path/to/zope.interface-3.x.x.tar.gz
$ cd zope.interface-3.x.x
$ python setup.py build
$ python setup.py install
```

Questi metodi installano la ZCA sul Python di *sistema*, nella cartella `site-packages`, ma questo potrebbe creare problemi. In un post sulla mailing list di Zope3, Jim Fulton sconsiglia l'utilizzo del Python di sistema [10]. In alternativa si può utilizzare `virtualenv` e/o `zc.buildout` per installare qualsiasi pacchetto Python. Questo metodo è adatto anche per il deploy.

1.4 Come provare il codice

In Python ci sono due approcci per la configurazione di ambienti di lavoro isolati per lo sviluppo di applicazioni. Il primo è *virtualenv* creato da Ian Biking e l'altro è *zc.buildout* creato da Jim Fulton. È anche possibile utilizzare questi due pacchetti insieme. Con questi pacchetti è possibile installare `zope.component` e le altre dipendenze in un ambiente di lavoro isolato. Queste sono le buone pratiche per la sperimentazione di codice Python, e familiarizzare con questi strumenti tornerà utile quando si vorrà sviluppare e fare il deploy delle proprie applicazioni.

1.4.1 virtualenv

Si può installare *virtualenv* utilizzando *easy_install*:

```
$ easy_install virtualenv
```

Poi si può creare un nuovo ambiente in questo modo:

```
$ virtualenv --no-site-packages myve
```

Questo comando crea un nuovo ambiente virtuale nella cartella *myve*. Ora, dall'interno della cartella myve, è possibile installare `zope.component` e le sue dipendenze utilizzando il comando *easy_install* che si trova dentro alla cartella *myve/bin*:

```
$ cd myve
$ ./bin/easy_install zope.component
```

Ora è possibile importare `zope.interface` e `zope.component` dal vostro nuovo interprete *python* disponibile dentro alla cartella *myve/bin*:

```
$ ./bin/python
```

Questo comando fornisce un prompt Python che può essere utilizzato per eseguire il codice di questo libro.

[10] http://article.gmane.org/gmane.comp.web.zope.zope3/21045

1.4.2 zc.buildout

Utilizzando `zc.buildout` con la ricetta `zc.recipe.egg`, è possibile creare un interprete Python che ha a disposizione gli eggs specificati. Per prima cosa, installare `zc.buildout` utilizzando il comando *easy_install*. (è possibile farlo anche dentro all'ambiente virtuale). Per creare un nuovo buildout per fare esperimenti con gli egg Python, per prima cosa creare una cartella e inizializzarla utilizzando il comando *buildout init* :

```
$ mkdir mybuildout
$ cd mybuildout
$ buildout init
```

Ora la nuova cartella *buildout* è un buildout. Il file di configurazione di default per il buildout è *buildout.cfg*. Dopo l'inizializzazione, avrà questo contenuto:

```
[buildout]
parts =
```

Cambiamolo così:

```
[buildout]
parts = py

[py]
recipe = zc.recipe.egg
interpreter = python
eggs = zope.component
```

Ora lanciamo il comando *buildout* disponibile dentro alla cartella *mybuildout/bin* senza argomenti. Questo crea un nuovo interprete Python dentro alla cartella *mybuildout/bin*:

```
$ ./bin/buildout
$ ./bin/python
```

Questo comando farà apparire un prompt Python che può essere utilizzato per eseguire il codice di questo libro.

CHAPTER 2

Un esempio

2.1 Introduzione

Consideriamo come esempio un'applicazione commerciale per la registrazione degli ospiti di un hotel. Python può implementare questa applicazione in vari modi. Inizieremo dando una breve occhiata ad una possibile implementazione procedurale, e poi ci sposteremo verso un semplice approccio orientato agli oggetti. Mentre esamineremo l'approccio orientato agli oggetti, vedremo come potremo trarre beneficio dai pattern di design classici, *adapter* e *interface*. Questo ci porterà nel mondo della Zope Component Architecture.

2.2 Approccio procedurale

In qualsiasi applicazione commerciale, una delle parti principali è la conservazione dei dati. Per semplicità, in questo esempio utilizzeremo un dizionario Python come sistema di storage. Creeremo degli id univoci per il dizionario e il valore associato ad ogni chiave sarà a sua volta un dizionario con i dettagli della prenotazione.

```
>>> bookings_db = {} #key: unique Id, value: details in a dictionary
```

Un'implementazione minimale richiede una funzione che verifichi i dettagli della prenotazione e una funzione di supporto che fornisca gli id univoci per le chiavi del dizionario di storage.

Possiamo generare un id univoco in questo modo:

```
>>> def get_next_id():
...     db_keys = bookings_db.keys()
...     if db_keys == []:
...         next_id = 1
...     else:
...         next_id = max(db_keys) + 1
...     return next_id
```

Come si può notare, l'implementazione della funzione *get_next_id* è molto semplice. La funzione prende una lista di chiavi e controlla una lista vuota. Se la lista è vuota questa è la nostra prima prenotazione, quindi restituiamo *1*. Se la lista non è vuota, aggiungiamo *1* al valore massimo della lista e lo restituiamo.

Ora utilizzeremo la funzione sopra per inserire degli elementi nel dizionario *bookings_db*:

```
>>> def book_room(name, place):
...     next_id = get_next_id()
...     bookings_db[next_id] = {
...     'name': name,
```

```
...     'room': place
...     }
```

Un'applicazione per la gestione delle prenotazioni di un hotel ha bisogno di dati supplementari:

- numero di telefono
- opzioni della camera
- metodo di pagamento
- ...

e ha bisogno di codice per la gestione dei dati:

- cancellare una prenotazione
- aggiornare una prenotazione
- pagare una stanza
- rendere i dati persistenti
- assicurare la sicurezza dei dati
- ...

Se dovessimo continuare con l'esempio procedurale, dovremmo creare molte funzioni e dovremmo passare i dati avanti e indietro tra di loro. Man mano che i requisiti cambiano o aumentano, il codice diventa sempre più difficile da manutenere e i bug diventano più difficili da correggere.

Possiamo terminare qui la nostra discussione sull'approccio procedurale poichè sarà molto più facile fornire la persistenza dei dati, la flessibilità di design e la testabilità del codice utilizzando gli oggetti.

2.3 Approccio orientato agli oggetti

La nostra discussione sul design orientato agli oggetti ci porta a introdurre la *classe*. La *classe* serve ad incapsulare i dati e il codice per gestirli.

La classe principale sarà il *FrontDesk*. La classe FrontDesk o verso cui delegherà la gestione, saprà come gestire i dati dell'hotel. Andremo a creare delle *istanze* di FrontDesk per applicare questa conoscenza al mestiere di gestire un hotel.

L'esperienza ha mostrato che incapsulare il codice e i dati attraverso gli oggetti porta ad un design più facile da comprendere,testare e modificare.

Vediamo i dettagli dell'implementazione di una classe FrontDesk:

```
>>> class FrontDesk(object):
...
...     def book_room(self, name, place):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': name,
...         'place': place
...         }
```

In questa implementazione, l'oggetto *frontdesk* (istanza della classe *FrontDesk*) è in grado di gestire le prenotazioni. Possiamo usare questa classe così:

```
>>> frontdesk = FrontDesk()
>>> frontdesk.book_room("Jack", "Bangalore")
```

Qualsiasi progetto reale sarà soggetto a cambiamenti nei requisiti. In questo caso la gestione dell'hotel ha deciso che ogni ospite deve fornire anche un numero di telefono, quindi siamo costretti a cambiare il codice.

Possiamo raggiungere questo requisito aggiungendo un argomento al metodo *book_room* che verrà aggiunto al dizionario dei valori:

```
>>> class FrontDesk(object):
...
...     def book_room(self, name, place, phone):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': name,
...         'place': place,
...         'phone': phone
...         }
```

Oltre a migrare i dati verso il nuovo schema, ora dobbiamo anche cambiare le chiamate a FrontDesk. Se però noi astraiamo i dettagli dell'ospite in un oggetto e lo usiamo per la registrazione, i cambiamenti al codice vengono minimizzati. Così possiamo applicare i cambiamenti ai dettagli dell'ospite e le chiamate a FrontDesk non avranno bisogno di cambiamenti.

Così abbiamo:

```
>>> class FrontDesk(object):
...
...     def book_room(self, guest):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }
```

Dobbiamo ancora cambiare il codice per rispondere ai cambiamenti dei requisiti. Sebbene questo sia inevitabile, il nostro obiettivo è quello di minimizzare questi cambiamenti, in modo da aumentare la manutenibilità.

Note: Quando si aggiunge del codice, è importante sentirsi liberi di apportare i cambiamenti senza paura di rompere l'applicazione. Il modo per avere i riscontri richiesti immediatamente è usare i test automatizzati. Con dei test ben scritti (e un buon sistema di controllo di versione) è possibile fare cambiamenti piccoli o grandi senza conseguenze. Una buona fonte di informazioni sulla filosofia della programmazione è il libro *Extreme Programming Explained* di Kent Beck.

Con l'introduzione dell'oggetto *ospite*, abbiamo risparmiato un pò di scrittura di codice e , cosa ancora più importante, l'astrazione fornita dall'oggetto ospite ha reso il sistema più semplice e più comprensibile. Come risultato, il codice è più facile da ri-fattorizzare e da mantenere.

2.4 Il pattern adapter

Nelle applicazioni reali, l'oggetto frontdesk dovrebbe eseguire compiti come la cancellazione e l'aggiornamento delle prenotazioni. Nel design attuale dobbiamo passare l'oggetto ospite al frontdesk ogni volta che chiamiamo metodi come *cancel_booking* e *update_booking*.

Possiamo evitare facilmente questo vincolo se passiamo l'oggetto *ospite* al metodo `FrontDesk.__init__()`, rendendolo così un attributo dell'istanza:

```
>>> class FrontDeskNG(object):
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def book_room(self):
...         guest = self.guest
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }
```

In effetti la soluzione che abbiamo raggiunto è un pattern molto conosciuto, l'*adapter* (adattatore). In generale, un adapter *contiene* un oggetto *adattato*:

```
>>> class Adapter(object):
...
...     def __init__(self, adaptee):
...         self.adaptee = adaptee
```

Questo pattern sarà utile quando si avrà a che fare con i dettagli implementativi che dipendono da considerazioni riguardanti:

- il cambio dei requisiti del cliente
- requisiti di persistenza dei dati (ZODB, RDBMS, XML...)
- requisiti di output (HTML, PDF, testo semplice...)
- il linguaggio di markup usato per il rendering (ReST, Markdown, Textile...)

Grazie agli adapters e al *component registry* (registro dei componenti), la ZCA permette di cambiare i dettagli implementativi del codice attraverso la *configurazione*.

Come vedremo in questa sezione sugli adapter della ZCA, la possibilità di configurare i dettagli implementativi fornisce utili abilità:

- l'abilità di passare da una implementazione all'altra
- l'abilità di aggiungere implementazioni quando necessario
- aumenta il riutilizzo sia del codice precedente sia del codice della ZCA

Queste capacità portano il codice ad essere più flessibile, scalabile e riutilizzabile. Tuttavia c'è un costo per tutto ciò poiché il mantenimento del *component registry* aggiunge un livello di complessità all'applicazione. Se è noto a priori che un'applicazione non avrà mai bisogno di queste funzionalità, la ZCA non è necessaria.

Ora siamo pronti per iniziare il nostro studio della *Zope Component Architecture*, iniziando dalle interfacce.

CHAPTER 3

Interfacce

3.1 Introduzione

Il file README.txt [11] nel percorso *path/to/zope/interface* definisce le interfacce in questo modo

```
Le interfacce sono oggetti che specificano (documentano) il comportamento
verso l'esterno degli oggetti che le "forniscono". Un'interfaccia specifica
il suo comportamento attraverso:

- la documentazione informale in una doc string.

- la definizione degli attributi

- le Invariants (invarianti), sono condizioni che devono essere verificate
per un oggetto che fornisce l'interfaccia.
```

Il libro classico dell'ingegneria del software «Design Patterns» [12] della *Gang of Four* raccomanda di "Programmare per interfacce, non per implementazione". Definire un'interfaccia formale è utile per la comprensione del sistema. In più, le interfacce portano a tutti i benefici della ZCA.

Un'interfaccia specifica le caratteristiche di un oggetto, il suo comportamento, le sue capacità. L'interfaccia descrive *cosa* può fare un oggetto, mentre per capire *come* lo fa, si dovrà guardare l'implementazione.

Due metafore usate comunemente per spiegare le interfacce sono i *contratti* e le *cianografie*, termini dei dizionari legale e architetturale per indicare un insieme di specifiche.

In alcuni linguaggi moderni come il Java, C#, VB.NET etc. le interfacce sono un aspetto esplicito del linguaggio. Siccome in Python mancano le interfacce, la ZCA le implementa con delle meta-classi da cui ereditare.

Di seguito un classico esempio di *hello world*:

```
>>> class Host(object):
...
...     def goodmorning(self, name):
...         """Say good morning to guests"""
...
...         return "Good morning, %s!" % name
```

Nel classe qui sopra abbiamo definito un metodo *goodmorning*. Se chiamiamo il metodo *goodmorning* su un oggetto istanza di questa classe, esso resituirà *Good morning, ...!*

[11] L'albero del codice di Zope è pieno di file README.txt che offrono una meravigliosa documentazione.
[12] http://en.wikipedia.org/wiki/Design_Patterns

```
>>> host = Host()
>>> host.goodmorning('Jack')
'Good morning, Jack!'
```

Qui *host* indica l'oggetto attuale utilizzato dal codice. Se si volesse esaminare i dettagli implementativi si dovrebbe accedere alla classe *Host*, o attraverso il codice sorgente o con uno strumento di documentazione delle API [13].

Ora inizieremo ad utilizzare le interfacce della ZCA. Per la classe sopra si può specificare l'interfaccia così:

```
>>> from zope.interface import Interface

>>> class IHost(Interface):
...
...     def goodmorning(guest):
...         """Say good morning to guest"""
```

Come si può notare, l'interfaccia eredita da `zope.interface.Interface`. Questo utilizzo (abuso?) dello statement *class* del Python è come la ZCA definisce le interfacce. Il prefisso "I" per i nomi delle interfacce non è altro che un'utile convenzione.

3.2 Dichiarazione delle interfacce

Abbiamo già visto come dichiarare un interfaccia utilizzando `zope.interface` nella sezione precedente. Questa sezione spiegherà il concetto più nel dettaglio.

Si consideri questa interfaccia di esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute

>>> class IHost(Interface):
...     """A host object"""
...
...     name = Attribute("""Name of host""")
...
...     def goodmorning(guest):
...         """Say good morning to guest"""
```

L'interfaccia *IHost* ha due attributi, *name* e *goodmorning*. Si ricordi che, in Python, i metodi sono anche attributi delle classi. L'attributo *name* è definito utilizzando la classe `zope.interface.Attribute`. Quando si aggiunge un attributo *name* all'interfaccia *IHost*, non viene impostato un valore iniziale. Lo scopo di definire l'attributo *name* qui è puramente per indicare che qualsiasi implementazione di questa interfaccia dovrà fornire un attributo chiamato *name*. In questo caso, non viene nemmeno indicato di che tipo deve essere l'attributo! Si può passare una stringa di documentazione come primo argomento di *Attribute*.

L'altro attributo, *goodmorning*, è un metodo definito utilizzando la definizione di funzione. Si noti che *self* non è richiesto nelle interfacce, perché *self* è un dettaglio implementativo della classe. Ad esempio, un modulo potrebbe implementare questa interfaccia. Se un modulo implementa questa interfaccia, saranno definiti al suo interno un attributo *name* e una funzione *goodmorning* e la funzione *goodmorning* accetterà un argomento.

Ora vedremo come fare la connessione interfaccia-classe-oggetto. Gli oggetti sono la vera parte attiva e sono istanze delle classi. L'interfaccia è la vera definizione dell'oggetto, quindi la classe è solo un dettaglio implementativo. Ecco perché si dovrebbe sempre programmare un'interfaccia e non un'implementazione.

Ora si dovrebbe prendere familiarità con due ulteriori termini per comprendere altri concetti. Il primo è *provide* (fornisce) e l'altro è *implement* (implementa). Gli oggetti **forniscono** le interfacce e le classi **implementano** le interfacce.

[13] http://en.wikipedia.org/wiki/Application_programming_interface

In altre parole, gli oggetti forniscono le interfacce che le loro classi implementano. Nel esempio sopra *host* (l'oggetto) fornisce *IHost* (l'interfaccia) e *Host* (la classe) implementa *IHost* (l'interfaccia). Un oggetto può fornire più di una interfaccia e anche una classe può implementare più di una interfaccia. Gli oggetti possono anche fornire delle interfacce direttamente in aggiunta alle interfacce implementate dalle loro classi.

Note: Le classi sono i dettagli implementativi degli oggetti. In Python, le classi sono oggetti chiamabili, quindi perché altri oggetti chiamabili non possono implementare un'interfaccia? In effetti possono. Per qualsiasi *oggetto chiamabile* è possibile dichiarare che esso produce oggetti che forniscono una qualche interfaccia dichiarando che l'*oggetto chiamabile* implementa le interfacce. Gli *oggetti chiamabili* sono generalmente chiamati *factories* (fabbriche). Dato che le funzioni sono oggetti chiamabili, una funzione può essere un *implementatore* di una interfaccia.

3.3 Implementare le interfacce

Per dichiarare che una classe implementa una particolare interfaccia, si utilizza la funzione `zope.interface.implements` nella definizione della classe.

Si consideri questo esempio, qui *Host* implementa *IHost*:

```
>>> from zope.interface import implements

>>> class Host(object):
...
...     implements(IHost)
...
...     name = u''
...
...     def goodmorning(self, guest):
...         """Say good morning to guest"""
...
...         return "Good morning, %s!" % guest
```

Note: se ci si chiede come lavori la funzione *implements*, si faccia riferimento al post del blog di James Henstridge (http://blogs.gnome.org/jamesh/2005/09/08/python-class-advisors/). Nella sezione degli adapter, si potrà vedrà la funzione *adapts*, che lavora in maniera simile.

Siccome *Host* implementa *IHost*, le istanze di *Host* forniscono *IHost*. C'è qualche metodo di utilità per introspezionare le dichiarazioni. La dichiarazione può essere fatta anche fuori dalla classe. Se si omette *interface.implements(IHost)* nel esempio sopra, una volta che la classe è già stata definita, è possibile scrivere:

```
>>> from zope.interface import classImplements
>>> classImplements(Host, IHost)
```

3.4 Esempio rivisitato

Ora, ritorniamo all'applicazione di esempio. Qui si vedrà come definire l'interfaccia dell'oggetto frontdesk:

```
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
```

```
...
...     def register():
...         """Register object's details"""
...
```

Per prima cosa abbiamo importato la classe *Interface* dal modulo `zope.interface`. Se si definisce una sottoclasse della classe *Interface* essa sarà una interfaccia dal punto di vista della Zope component architecture. Un'interfaccia può essere implementata, come abbiamo già visto, in una classe o in qualsiasi oggetto chiamabile.

L'interfaccia frontdesk definita qui è *IDesk*. La stringa di documentazione dell'interfaccia fornisce un'idea di un possibile oggetto. Nella definizione di un metodo in un'interfaccia, il primo argomento **non** deve essere *self*, poiché un'interfaccia non verrà mai istanziata e i suoi metodi non saranno mai chiamati. Al contrario, la classe interfaccia documenta semplicemente come dovrebbero apparire i metodi e gli attributi in qualsiasi classe normale che dichiari di implementarla, e il parametro *self* è un dettaglio implementativo che non ha bisogno di essere documentato.

Come sappiamo, un'interfaccia può anche specificare normali attributi:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute

>>> class IGuest(Interface):
...
...     name = Attribute("Name of guest")
...     place = Attribute("Place of guest")
```

In questa interfaccia, l'oggetto ospite ha due attributi specificati con la documentazione. Un'interfaccia può anche specificare attributi e metodi insieme. Un'interfaccia può essere implementata da una classe, da un modulo o qualsiasi altro oggetto. Per esempio una funzione può creare dinamicamente un componente e restituirlo; in questo caso la funzione è un implementatore dell'interfaccia.

Ora sappiamo cos'è un'interfaccia e come definirla e usarla. Nel prossimo capitolo vedremo come utilizzare un'interfaccia per definire un componente adapter.

3.5 Interfacce marker

Un'interfaccia può essere utilizzata per dichiarare che un particolare oggetto appartiene ad uno speciale tipo. Un'interfaccia senza attributi o metodi è chiamata *interfaccia marker*.

Ecco un esempio di *interfaccia marker*:

```
>>> from zope.interface import Interface

>>> class ISpecialGuest(Interface):
...     """A special guest"""
```

Questa interfaccia può essere utilizzata per indicare che un oggetto è uno speciale tipo di ospite.

3.6 Invarianti

A volte c'è la necessità di utilizzare alcune regole per un componente che coinvolgono uno o più normali attributi. Questo tipo di regole sono chiamate *invariants* (invarianti). Si può utilizzare `zope.interface.invariant` per impostare delle *invarianti* sulle interfacce degli oggetti.

Si consideri un semplice esempio: c'è un oggetto *persona* con gli attributi *name*,'email' e *phone*. Come si potrebbe implementare una regola di validazione che imponga che almeno uno fra gli attributi *email* e *phone* debba esistere ma non necessariamente entrambi?

Per prima cosa bisogna costruire un oggetto chiamabile, o una semplice funzione o una istanza chiamabile di una classe come questa:

```
>>> def contacts_invariant(obj):
...
...     if not (obj.email or obj.phone):
...         raise Exception(
...             "At least one contact info is required")
```

Poi si deve definire l'interfaccia dell oggetto *person* in questo modo. Utilizzare la funzione `zope.interface.invariant` per definire l'invariante:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import invariant

>>> class IPerson(Interface):
...
...     name = Attribute("Name")
...     email = Attribute("Email Address")
...     phone = Attribute("Phone Number")
...
...     invariant(contacts_invariant)
```

Ora utilizzare il metodo *validateInvariants* dell'interfaccia per la validazione:

```
>>> from zope.interface import implements

>>> class Person(object):
...     implements(IPerson)
...
...     name = None
...     email = None
...     phone = None

>>> jack = Person()
>>> jack.email = u"jack@some.address.com"
>>> IPerson.validateInvariants(jack)
>>> jill = Person()
>>> IPerson.validateInvariants(jill)
Traceback (most recent call last):
...
Exception: At least one contact info is required
```

Come si può vedere l'oggetto *jack* è validato senza alcuna eccezione mentre l'oggetto *jill* non è stato validato dal vincolo invariante, così viene sollevata un'eccezione.

CHAPTER 4

Adapters

4.1 Implementazione

In questa sezione verranno descritti gli adapter in dettaglio. La Zope Component Architecture, come abbiamo già visto, aiuta ad utilizzare efficacemente gli oggetti Python. I componenti adapter sono uno dei componenti di base utilizzati dalla ZCA. Gli adapter sono oggetti Python, ma con interfacce ben definite.

Per dichiarare che una classe è un adapter si utilizza la funzione *adapts* definita nel pacchetto `zope.component`. Ecco il nuovo adattatore *FrontDeskNG* con una dichiarazione esplicita di interfaccia:

```
>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         guest = self.guest
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }
```

Quello che abbiamo definito qui è un *adapter* per *IDesk*, che adatta gli oggetti *IGuest*. L'interfaccia *IDesk* è implementata dalla classe *FrontDeskNG*. Quindi un'istanza di questa classe fornirà l'interfaccia *IDesk*

```
>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)
```

```
>>> IDesk.providedBy(jack_frontdesk)
True
```

Il *FrontDeskNG* è solo uno dei possibili adattatori. È possibile creare anche altri adapter che permettano di gestire le registrazioni degli ospiti diversamente.

4.2 Registration

Per utilizzare questo componente adapter, bisogna registrarlo nel *component registry* anche conosciuto come *site manager*. Un site manager normalmente risiede in un sito. Il sito e il suo site manager saranno più importanti quando si svilupperanno applicazioni Zope3. Per ora ci interesseremo solo del *global site* e del *global site manager* (o component registry). Il global site manager risiede in memoria mentre un local site manager è persistente.

Per registrare il nostro componente, per prima cosa recuperiamo il global site manager:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(FrontDeskNG,
...                     (IGuest,), IDesk, 'ng')
```

Per recuperare il global site manager, bisogna chiamare la funzione `getGlobalSiteManager` disponibile nel pacchetto `zope.component`. In effetti, il global site manager è disponibile anche come un attributo (*globalSiteManager*) del pacchetto `zope.component`. Quindi è anche possibile utilizzare direttamente l'attributo `zope.component.globalSiteManager`. Per registrare l'adapter nei componenti, come si può vedere sopra, si utilizza il metodo *registerAdapter* del component registry. Il primo argomento deve essere un classe/factory adapter. Il secondo argomento è una tupla di oggetti *adattati*, ad esempio l'oggetto che stiamo adattando. In questo esempio, stiamo adattando solo l'oggetto *IGuest*. Il terzo argomento è l'interfaccia implementata dal componente adater. Il quarto argomento è opzionale ed è il nome di quel particolare adapter. Dato che abbiamo dato un nome a questo adapter, questo è un *named adapter*. Se non viene passato alcun nome allora questo sarà automaticamente una stringa vuota (‘’).

Nella registrazione sopra abbiamo passato l'interfaccia adattata e l'interfaccia fornita dall'adapter. Dato che questi dettagli sono già stati specificati nella implementazione dell'adapter, non è necessario specificarli ancora. Infatti, avremmo potuto fare la registrazione così

```
>>> gsm.registerAdapter(FrontDeskNG, name='ng')
```

Ci sono alcune vecchie API per fare la registrazione che però andrebbero evitate. Le funzioni delle vecchie API iniziano con *provide*, ad es. *provideAdapter*, *provideUtility*,etc. Durante lo sviluppo di un'applicazione Zope3 è possibile utilizzare lo Zope configuration markup language (ZCML) per la registrazione dei componenti. In Zope3, i *local component* (o componenti persistenti) possono essere registrati dalla Zope Management Interface (ZMI) o anche in maniera programmatica.

Note: I local component sono componenti persistenti mentre i global component risiedono in memoria. I global component saranno registrati in base alla configurazione dell'applicazione. I local component sono caricati in memoria dal database all'avvio dell'applicazione.

4.3 Recuperare un adapter

Il recupero dei componenti registrati dal component registry può essere effettuato con due funzioni disponibili nel pacchetto `zope.component`. Una di esse è *getAdapter* e l'altra è *queryAdapter*. Entrambe le funzioni accettano gli

stessi parametri. Il metodo *getAdapter* solleverà *ComponentLookupError* se la ricerca del componente fallisce, mentre *queryAdapter* restituirà *None*.

Si possono importare i due metodi in questo modo:

```
>>> from zope.component import getAdapter
>>> from zope.component import queryAdapter
```

Nella sezione precedente abbiamo registrato un componente per l'oggetto ospite (l'oggetto adattato) che fornisce l'interfaccia *IDesk* con nome 'ng'. Nella prima sezione di questo capitolo, abbiamo creato un oggetto ospite di nome *jack*.

Ecco come recuperare un componente che adatta l'interfaccia dell'oggetto *jack* (*IGuest*) e fornisce l'interfaccia *IDesk* e con il nome 'ng'. Qui sia *getAdapter* sia *queryAdapter* lavorano in maniera simile:

```
>>> getAdapter(jack, IDesk, 'ng')
<FrontDeskNG object at ...>
>>> queryAdapter(jack, IDesk, 'ng')
<FrontDeskNG object at ...>
```

Come si può vedere il primo argomento è l'oggetto da adattare, poi l'interfaccia che dovrebbe essere fornita dal componente e per ultimo il nome del componente adapter.

Se si prova a cercare un componente con un nome non registrato ma per lo stesso oggetto adattato e la stessa interfaccia, la ricerca fallirà. Ecco come si comportano i due metodi in questo caso:

Come si può vedere sopra, *getAdapter* ha sollevato un'eccezione *ComponentLookupError* mentre *queryAdapter* ha restituito *None* quando la ricerca è fallita.

Il terzo argomento, il nome di registrazione, è opzionale e se non viene passato il suo valore predefinito sarà una stringa vuota (''). Dal momento che non ci sono componenti registrati con una stringa vuota, *getAdapter* solleverà *ComponentLookupError* e 'queryAdapter' restituirà *None*:

```
>>> getAdapter(jack, IDesk)
Traceback (most recent call last):
...
ComponentLookupError: ...
>>> reg = queryAdapter(jack, IDesk)
>>> reg is None
True
```

In questa sezione abbiamo imparato come registrare un semplice adapter e come recuperarlo dal component registry. Questo tipo di adapter sono chiamati single adapter (adattatore singolo) perché adattano solo un oggetto. Se un adapter adatta più di un oggetto, allora si chiama multi-adapter (multi-adattatore).

4.4 Recuperare gli adapter tramite le interfacce

Gli adapter possono essere recuperati direttamente utilizzando le interfacce, ma questo funziona solo per gli adapter senza nome. Il primo argomento è l'oggetto adattato e il secondo è un argomento keyword. Se la ricerca dell'adapter fallisce, viene restituito il secondo argomento.

```
>>> IDesk(jack, alternate='default-output')
'default-output'
```

Il nome della keyword può anche essere ommesso:

```
>>> IDesk(jack, 'default-output')
'default-output'
```

Se il secondo argomento non viene passato allora viene sollevata *TypeError*:

```
>>> IDesk(jack)
Traceback (most recent call last):
...
TypeError: ('Could not adapt',
  <Guest object at ...>,
  <InterfaceClass __builtin__.IDesk>)
```

Qui *FrontDeskNG* viene registrato senza nome:

```
>>> gsm.registerAdapter(FrontDeskNG)
```

Ora la ricerca dell'adapter dovrebbe andare a buon fine:

```
>>> IDesk(jack, 'default-output')
<FrontDeskNG object at ...>
```

Quindi, per casi semplici, si può utilizzare l'interfaccia per recuperare il componente adapter.

4.5 Il pattern adapter

Il concetto di adapter nella Zope Component Architecture è molto simile al classico *pattern adapter* che viene descritto nel libro «Design Pattern». L'intento degli adapter della ZCA è però più ampio di quello del *pattern adapter*. L'intento del *pattern adapter* è quello di convertire l'interfaccia di una classe in un'altra interfaccia che il client si aspetta. Questo permette di poter far lavorare insieme le classi che altrimenti sarebbero incompatibili a causa delle loro interfacce. Ma nella sezione *Motivation* del libro «Design Pattern», GoF dice: "Spesso l'adapter fornisce delle funzionalità che le classi adattate non forniscono". L'adapter della ZCA è più incentrato sull'aggiunta di funzionalità che sulla creazione di una nuova interfaccia per un oggetto adattato. L'adapter della ZCA permette alle classi adapter di estendere le funzionalità aggiungendo nuovi metodi. (sarebbe interessante notare che l'*Adapter* era conosciuto come *Feature* nelle prime fasi del design della ZCA.) [14]

Nel paragrafo sopra c'è una citazione dal libro della "Gang of Four", che finisce così "...che le **classi** adattate non forniscono". Ma nella frase successiva io ho utilizzato "oggetto adattato" invece di "classe adattata", poiché Gof descrive due varianti di adapter basati sull'implementazione. La prima è chiamata *class adapter* e l'altra *object adapter*. Un class adapter utilizza l'ereditarietà multipla per adattare un'interfaccia all'altra, mentre un object adapter fa affidamento sulla composizione degli oggetti. L'adapter della ZCA segue il pattern object adapter, il quale usa la delega come meccanismo di composizione. Il secondo principio di GoF a proposito del design orientato agli oggetti dice: "Favorite la composizione degli oggetti rispetto all'ereditarietà di classe". Per maggiori dettagli su questo argomento vi invito a leggere il libro «Design Pattern».

La cosa più interessante degli adapter della ZCA sono le interfacce esplicite per i componenti e il component registry. I componenti adapter della ZCA vengono registrati nel component registry e recuperati dagli oggetti client utilizzando le interfacce e il nome quando richiesto.

[14] Discussione sulla rinomina delle Feature in Adapter: http://mail.zope.org/pipermail/zope3-dev/2001-December/000008.html

CHAPTER 5

Utility

5.1 Introduzione

Ora conosciamo i concetti di interfaccia, adapter e component registry. A volte però sarebbe utile poter registrare un oggetto che non adatta nulla. Connessioni a database, parse XML, oggetti che restituiscono Id univoci, etc. sono tutti esempi di questo tipo di oggetti. Questo tipo di componenti forniti dalla ZCA sono chiamati *utility*.

Le utility sono solo oggetti che forniscono un'interfaccia e che vengono ricercati per interfaccia e per nome. Questo approccio crea un *global registry* attraverso il quale le interfacce possono essere registrate e accedute da diverse parti della nostra applicazione, senza bisogno di passare le istanze avanti e indietro come parametri.

Non è però consigliabile registrare tutte le istanze di componenti in questo modo. Si dovrebbero registrare solo i componenti che si vuole rendere rimpiazzabili.

5.2 Semplici utility

Una utility può essere registrata con un nome o senza nome. Una utility registrata con un nome è chiamata *named utility*, e la vedremo nella prossima sezione. Prima di implementare l'utility, come solito, definiamo la sua interfaccia. Ecco un'interfaccia `IGreeter` ("salutatore"):

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...
...     def greet(name):
...         """Say hello"""
```

Come anche un adapter, una utility può avere più di una implementazione. Ecco una possibile implementazione della interfaccia sopra:

```
>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         return "Hello " + name
```

La vera utility sarà un'istanza di questa classe. Per utilizzare questa utility dobbiamo registrarla per poterla richiedere in seguito utilizzando l'API della ZCA. Possiamo registrare un'istanza di questa classe (*utility*) utilizzando `registerUtility`:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter)
```

In questo esempio abbiamo registrato l'utility che fornisce l'interfaccia `IGreeter`. Si può ricercare l'interfaccia sia con `queryUtility` sia con `getUtility`:

```
>>> from zope.component import queryUtility
>>> from zope.component import getUtility

>>> queryUtility(IGreeter).greet('Jack')
'Hello Jack'

>>> getUtility(IGreeter).greet('Jack')
'Hello Jack'
```

Come si può vedere, gli adapter normalmente sono delle classi mentre le utility normalmente sono istanze di classi. L'istanza della classe utility viene creata solo una volta mentre le istanze dell'adapter vengono create dinamicamente quando vengono richieste.

5.3 Named utility

Quando si registra un componente, come ad esempio un adapter, è possibile assegnargli un nome. Come detto nella precedente sezione, una utility registrata con un particolare nome è chiamata *named utility*.

Ecco come registrare l'utility `greeter` con un nome:

```
>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter, 'new')
```

In questo esempio abbiamo registrato l'utility con un nome fornendo l'interfaccia `IGreeter`. Ecco come ricercare l'interfaccia con `queryUtility` o con `getUtility`:

```
>>> from zope.component import queryUtility
>>> from zope.component import getUtility

>>> queryUtility(IGreeter, 'new').greet('Jill')
'Hello Jill'

>>> getUtility(IGreeter, 'new').greet('Jill')
'Hello Jill'
```

Come si può vedere qui, quando si fa un'interrogazione è necessario utilizzare il *name* come secondo argomento.

Chiamare la funzione `getUtility` senza un nome (come secondo argomento) è uguale a chiamare a chiamarla con una stringa vuota come nome poichè il valore predefinito del secondo argomento (keyword) è una stringa vuota. Poi il meccanismo di ricerca dei componenti proverà a trovare il componente il nome uguale alla stringa vuota e fallirà. Quando la ricerca del componente fallisce solleva l'eccezione `ComponentLookupError`. Si ricordi che non ritornerà un componente a caso con un'altro nome. Le funzioni di ricerca degli adapter, `getAdapter` e `queryAdapter` lavorano in maniera simile.

5.4 Factory

Una `factory` è un componente utility che fornisce l'interfaccia `IFactory`.

Per creare una factory, per prima cosa definiamo l'interfaccia dell'oggetto:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IDatabase(Interface):
...
...     def getConnection():
...         """Return connection object"""
```

Ecco una finta implementazione dell'interfaccia `IDatabase`:

```
>>> class FakeDb(object):
...
...     implements(IDatabase)
...
...     def getConnection(self):
...         return "connection"
```

Possiamo creare una factory utilizzando `zope.component.factory.Factory`:

```
>>> from zope.component.factory import Factory

>>> factory = Factory(FakeDb, 'FakeDb')
```

Ora possiamo registrarla in questo modo:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(factory, IFactory, 'fakedb')
```

Per utilizzare la factory, possiamo fare così:

```
>>> from zope.component import queryUtility
>>> queryUtility(IFactory, 'fakedb')()
<FakeDb object at ...>
```

C'è una scorciatoia per utilizzare una factory:

```
>>> from zope.component import createObject
>>> createObject('fakedb')
<FakeDb object at ...>
```

CHAPTER 6

Adapter avanzati

In questo capitolo discuteremo di adapter avanzati come i multi-adapter, i subscription adapter e gli handler.

6.1 Multi adapter

Un semplice adapter normalmente adatta solo un oggetto, ma un adapter può adattare più di un oggetto. Se un adapter adatta più di un oggetto, è chiamato *multi-adapter*.

```
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class IAdapteeOne(Interface):
...     pass

>>> class IAdapteeTwo(Interface):
...     pass

>>> class IFunctionality(Interface):
...     pass

>>> class MyFunctionality(object):
...     implements(IFunctionality)
...     adapts(IAdapteeOne, IAdapteeTwo)
...
...     def __init__(self, one, two):
...         self.one = one
...         self.two = two

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerAdapter(MyFunctionality)

>>> class One(object):
...     implements(IAdapteeOne)

>>> class Two(object):
...     implements(IAdapteeTwo)

>>> one = One()
>>> two = Two()
```

```
>>> from zope.component import getMultiAdapter

>>> getMultiAdapter((one,two), IFunctionality)
<MyFunctionality object at ...>

>>> myfunctionality = getMultiAdapter((one,two), IFunctionality)
>>> myfunctionality.one
<One object at ...>
>>> myfunctionality.two
<Two object at ...>
```

6.2 Subscription adapter

A differenza dei normali adapter, i *subscription adapter* vengono utilizzati quando vogliamo recuperare tutti gli adapter che adattano un oggetto a una particolare interfaccia. Un subscription adapter è anche conosciuto come *subscriber*.

Consideriamo un problema di validazione. Abbiamo degli oggetti e vogliamo verificare se essi aderiscono a qualche tipo di standard. Si definisce un'interfaccia di validazione:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IValidate(Interface):
...
...     def validate(ob):
...         """Determine whether the object is valid
...
...         Return a string describing a validation problem.
...         An empty string is returned to indicate that the
...         object is valid.
...         """
```

Magari abbiamo dei documenti:

```
>>> class IDocument(Interface):
...
...     summary = Attribute("Document summary")
...     body = Attribute("Document text")

>>> class Document(object):
...
...     implements(IDocument)
...
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body
```

Ora, potremmo voler specificare diverse regole di validazione per questi documenti. Per esempio, potremmo richiedere che la descrizione sia una linea singola:

```
>>> from zope.component import adapts

>>> class SingleLineSummary:
...
...     adapts(IDocument)
...     implements(IValidate)
```

```
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if '\n' in self.doc.summary:
...             return 'Summary should only have one line'
...         else:
...             return ''
```

Oppure potremmo richiedere che il corpo del testo sia lungo al massimo 1000 caratteri:

```
>>> class AdequateLength(object):
...
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if len(self.doc.body) < 1000:
...             return 'too short'
...         else:
...             return ''
```

Possiamo registrare queste regole come subscription adapter:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerSubscriptionAdapter(SingleLineSummary)
>>> gsm.registerSubscriptionAdapter(AdequateLength)
```

In seguito possiamo utilizzare i subscriber per validare gli oggetti:

```
>>> from zope.component import subscribers

>>> doc = Document("A\nDocument", "blah")
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['Summary should only have one line', 'too short']

>>> doc = Document("A\nDocument", "blah" * 1000)
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['Summary should only have one line']

>>> doc = Document("A Document", "blah")
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['too short']
```

6.3 Handler

Gli handler sono delle fabbriche di subscription adapter che non restituiscono nulla. Essi infatti eseguono tutto il loro lavoro quando vengono chiamati. Gli handler tipicamente sono utilizzati per la gestione degli eventi e sono anche conosciuti come *event subscribers* o *event subscription adapter*.

Gli event subscriber sono diversi dagli altri subscription adapter per il fatto che il chiamante dell'event subscriber non si aspetta di interagire con loro in nessun modo diretto. Per esempio, un generatore di eventi non si aspetta di ricevere alcun valore di ritorno. Poiché i subscribers non hanno bisogno di fornire alcuna API ai loro chiamanti, è più naturale definirli con delle funzioni, piuttosto che con delle classi. Per esempio, in un sistema di gestione documentale potremmo voler registrare le date di creazione dei documenti:

```
>>> import datetime

>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()
```

In questo esempio, abbiamo una funzione che prende un evento e svolge qualche operazione e in effetti non restituisce nulla. Questo è un caso speciale di subscription adapter che adatta un evento verso nulla. Tutto il lavoro è svolto quando la "factory" dell'adapter viene chiamata. I subscriber che non restituiscono niente sono chiamati "handler" e per registrarli ci sono delle API specifiche:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IDocumentCreated(Interface):
...
...     doc = Attribute("The document that was created")

>>> class DocumentCreated(object):
...
...     implements(IDocumentCreated)
...
...     def __init__(self, doc):
...         self.doc = doc
```

Dovremo anche cambiare la definizione del nostro handler:

Questo identifica l'handler come un adapter di eventi di tipo *IDocumentCreated*.

Andiamo a registrare l'handler:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerHandler(documentCreated)
```

Ora possiamo creare un evento e utilizzare la funzione *handle* per chiamare gli handler registrati per l'evento:

```
>>> from zope.component import handle

>>> handle(DocumentCreated(doc))
>>> doc.created.__class__.__name__
'datetime'
```

CHAPTER 7

Utilizzo della ZCA in Zope

La *Zope Component Architecture* viene utilizzata sia in Zope3 sia in Zope2. Questo capitolo tratterà l'utilizzo della ZCA in Zope.

7.1 ZCML

Lo **Zope Configuration Markup Language (ZCML)** è un sistema di configurazione basato su XML per la registrazione dei componenti. Così, invece di utilizzare le API Python per la registrazione, è possibile utilizzare lo ZCML. Sfortunatamente però l'utilizzo dello ZCML richiederà l'installazione di più pacchetti di dipendenze.

Per installare questi pacchetti lanciare:

```
$ easy_install "zope.component [zcml]"
```

Ecco come registrare un componente:

```
<configure xmlns="http://namespaces.zope.org/zope">

<adapter
    factory=".company.EmployeeSalary"
    provides=".interfaces.ISalary"
    for=".interfaces.IEmployee"
    />
```

Gli attributi *provides* e *for* sono opzionali, a patto che siano già stati dichiarati nell'implementazione del componente:

```
<configure xmlns="http://namespaces.zope.org/zope">

<adapter
    factory=".company.EmployeeSalary"
    />
```

Se si vuole registrare il componente come un named adapter, si può fornire un attributo *name*:

```
<configure xmlns="http://namespaces.zope.org/zope">

<adapter
    factory=".company.EmployeeSalary"
    name="salary"
    />
```

Anche le utility sono registrate in maniera simile:

```
<configure xmlns="http://namespaces.zope.org/zope">

<utility
    component=".database.connection"
    provides=".interfaces.IConnection"
    />
```

l'attributo *provides* è opzionale, a patto che sia stato dichiarato nell'implementazione:

```
<configure xmlns="http://namespaces.zope.org/zope">

<utility
    component=".database.connection"
    />
```

Se si vuole registrare il componente come named utility, si può fornire l'attributo *name*:

```
<configure xmlns="http://namespaces.zope.org/zope">

<utility
    component=".database.connection"
    name="Database Connection"
    />
```

Invece di utilizzare direttamente il componente, è possibile anche fornire la factory:

```
<configure xmlns="http://namespaces.zope.org/zope">

<utility
    factory=".database.Connection"
    />
```

7.2 Overrides

Quando registriamo un componente utilizzando le API Python (i metodi *register**), l'ultimo componente registrato rimpiazzerà il componente registrato in precedenza se entrambi sono registrati con gli stessi componenti. Per esempio, consideriamo l'esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IA(Interface):
...     pass

>>> class IP(Interface):
...     pass

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> class AP(object):
...
...     implements(IP)
...     adapts(IA)
```

```
...
...     def __init__(self, context):
...         self.context = context

>>> class AP2(object):
...
...     implements(IP)
...     adapts(IA)
...
...     def __init__(self, context):
...         self.context = context

>>> class A(object):
...
...     implements(IA)

>>> a = A()
>>> ap = AP(a)

>>> gsm.registerAdapter(AP)

>>> getAdapter(a, IP)
<AP object at ...>
```

Se registriamo un'altro adapter quello esistente viene rimpiazzato:

```
>>> gsm.registerAdapter(AP2)

>>> getAdapter(a, IP)
<AP2 object at ...>
```

Ma quando si registrano i componenti utilizzando ZCML, la seconda registrazione solleva un errore di conflitto. Questo è un suggerimento per noi, altrimenti ci sarebbe la possibilità di sovrascrivere le registrazioni per sbaglio e questo potrebbe portare a una maggiore difficoltà nel tracciare i bug nel sistema. Quindi l'utilizzo dello ZCML è una buona cosa per l'applicazione.

A volte avremo la necessità di sovrascrivere una registrazione esistente. Per questa evenienza lo ZCML fornisce la direttiva *includeOverrides*. Con questa direttiva possiamo scrivere le nostre sostituzioni in un file separato:

```
<includeOverrides file="overrides.zcml" />
```

7.3 NameChooser

Posizione: *zope.app.container.contained.NameChooser*

Questo è un adapter che permette di scegliere un nome univoco per un oggetto all'interno di un contenitore.

La registrazione dell'adapter è simile a questa:

```
<adapter
    provides=".interfaces.INameChooser"
    for="zope.app.container.interfaces.IWriteContainer"
    factory=".contained.NameChooser"
    />
```

Dalla registrazione possiamo vedere che l'oggetto adattato è un *IWriteContainer* e che l'adapter fornisce *INameChooser*.

Questo adapter fornisce una funzionalità molto comoda per i programmatori Zope. La principale implementazione di *IWriteContainer* in Zope3 sono *zope.app.container.BTreeContainer* e *zope.app.folder.Folder*. Normalmente erediteremo da queste implementazioni per creare le nostre classi contenitori. Se che non ci fosse nessuna interfaccia chiamata *INameChooser* e il relativo adapter, allora dovremmo implementare questa funzionalità per ogni implementazione separatamente.

7.4 LocationPhysicallyLocatable

Posizione: *zope.location.traversing.LocationPhysicallyLocatable*

Questo adapter viene utilizzato frequentemente nelle applicazioni Zope3 ma normalmente viene chiamato attraverso un API in *zope.traversing.api*. (Qualche vecchio codice utilizza le funzioni di `zope.app.zapi` che è solo una redirezione aggiuntiva)

La registrazione dell'adapter è simile a questa:

```
<adapter
    factory="zope.location.traversing.LocationPhysicallyLocatable"
    />
```

L'interfaccia fornita e l'interfaccia adattata sono specificate nell'implementazione.

Ecco qui l'inizio dell'implementazione:

```
class LocationPhysicallyLocatable(object):
    """Provide location information for location objects
    """
    zope.component.adapts(ILocation)
    zope.interface.implements(IPhysicallyLocatable)
    ...
```

Normalmente, quasi tutti gli oggetti persistenti nell'applicazione Zope3 forniranno l'interfaccia `ILocation`. Questa interfaccia ha solo due attributi, `__parent__` e `__name__`. Il `__parent__` è il contenitore nella gerarchia degli oggetti e `__name__` è il nome dell'oggetto all'interno del contenitore.

L'interfaccia *IPhysicallyLocatable* ha 4 metodi: `getRoot`, `getPath`, `getName`, e `getNearestSite`.

- `getRoot` restituisce l'oggetto radice fisica
- `getPath` restituisce il percorso fisico verso l'oggetto in formato stringa
- `getName` restituisce l'ultimo segmento del percorso fisico
- `getNearestSite` restituisce il sito in cui è contenuto l'oggetto. Se l'oggetto è un sito, viene restituito l'oggetto stesso.

Quando si studia Zope3, si capisce che queste sono le cose importanti e quelle che vengono richieste più spesso. Per comprendere la bellezza di questo sistema bisogna vedere come Zope2 recupera l'oggetto radice fisica e come questo è implementato. Esiste un metodo chiamato `getPhysicalRoot` virtualmente per ogni oggetto contenitore.

7.5 DefaultSized

Posizione: `zope.size.DefaultSized`

Questo adapter non è che l'implementazione di default dell'interfaccia `ISized`. Esso è registrato per tutti i tipi di oggetti. Se si vuole registrare questo adapter per una particolare interfaccia si dovrà sovrascrivere questa registrazione nella propria implementazione.

La registrazione dell'adapter è simile a questa:

```
<adapter
    for="*"
    factory="zope.size.DefaultSized"
    provides="zope.size.interfaces.ISized"
    permission="zope.View"
    />
```

Come si può vedere, l'interfaccia adattata è "*" quindi può adattare qualsiasi tipo di oggetto.

`ISized` è una semplice interfaccia con due contratti di metodi:

```
class ISized(Interface):

    def sizeForSorting():
        """Returns a tuple (basic_unit, amount)

        Used for sorting among different kinds of sized objects.
        'amount' need only be sortable among things that share the
        same basic unit."""

    def sizeForDisplay():
        """Returns a string giving the size.
        """
```

Si può trovare un'altro adapter `ISized` registrato per `IZPTPage` nel pacchetto `zope.app.zptpage`.

7.6 ZopeVersionUtility

Posizione: `zope.app.applicationcontrol.ZopeVersionUtility`

La registrazione è questa:

```
<utility
    component=".zopeversion.ZopeVersionUtility"
    provides=".interfaces.IZopeVersion" />
```

L'interfaccia fornita, `IZopeVersion`, ha solo un metodo chiamato `getZopeVersion`. Questo metodo restituisce una stringa contenente la versione di Zope (con eventualmente le informazione di SVN). L'implementazione di default, `ZopeVersionUtility`, prende le informazioni sull versione da un file *version.txt* nella cartella `zope/app`. Se Zope è in esecuzione a partire da un checkout di Subversion, esso mostra l'ultimo numero di revisione. Se nessuno dei metodi sopra funziona allora restituisce *Development/Unknown*.

CHAPTER 8

Caso di studio

Note: Questo capitolo non è ancora completo. Ogni suggerimento è benvenuto!

8.1 Introduzione

Questo capitolo è un esempio di creazione di un'applicazione desktop utilizzando la libreria PyGTK per le GUI e la ZCA. Quest'applicazione utilizza anche due diversi tipi di meccanismi per la persistenza dei dati, un database ad oggetti (ZODB) e un altro database relazionale (SQLite). In ogni caso nella pratica solo uno storage può essere utilizzato per una particolare installazione. La ragione di utilizzare due diversi meccanismi di persistenza è la dimostrazione di come usare la ZCA per incollare tra loro i componenti. La maggior parte del codice di questa applicazione è legato a PyGTK.

Man mano che l'applicazione crescie si potranno utilizzare i componenti ZCA dovunque si desideri avere modularità e estensibilità. Si utilizzino invece direttamente oggetti Python dove non sono richieste queste due proprietà.

Non c'è differenza nell'utilizzo della ZCA per il web o per il desktop o per qualsiasi altro tipo di applicazione o framework. È preferibile seguire una convenzione per posizione dalla quale registrare i componenti. Questa applicazione utilizza una convenzione che permette di essere estesa posizionando delle registrazioni di componenti simili in moduli separati e in seguito importarli dal modulo di registrazione principale. In questa applicazione il modulo principale per la registrazione dei componenti è `register.py`.

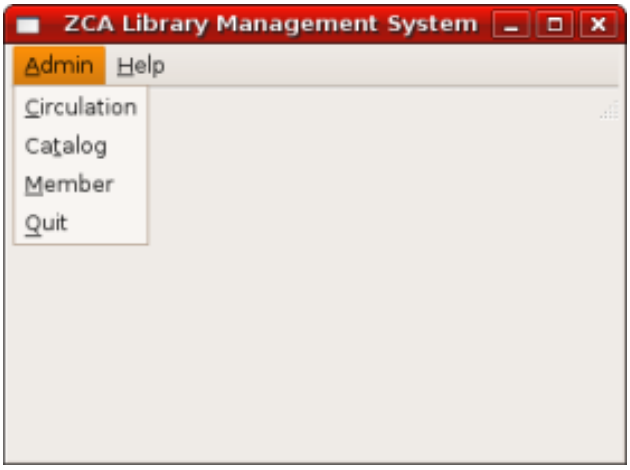
Il codice sorgente di questa applicazione può essere scaricato su http://www.muthukadan.net/downloads/zcalib.tar.bz2

8.2 Casi d'uso

L'applicazione che ora andiamo a discutere è un sistema per la gestione di una biblioteca con funzionalità minimali. I requisiti possono essere riassunti così:

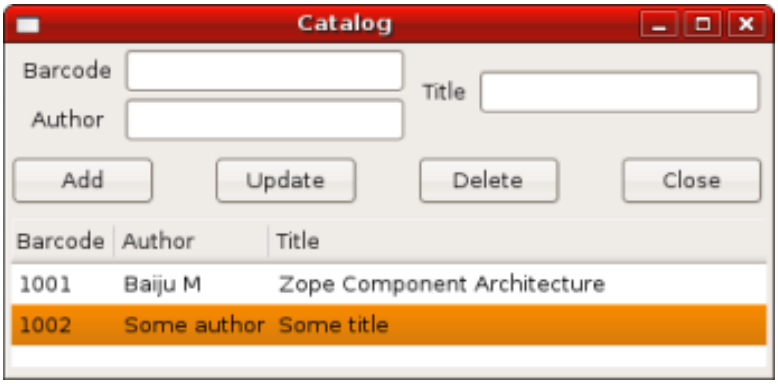
- aggiunta dei membri con un numero univoco e un nome
- aggiunta dei libri con il codice a barre, autore e titolo
- prestito dei libri
- restituzione dei libri

L'applicazione può essere disegnata in modo che le funzionalità principali possano essere utilizzate da una singola finestra. La finestra principale per accedere a tutte queste funzionalità potrebbe avere questo aspetto:

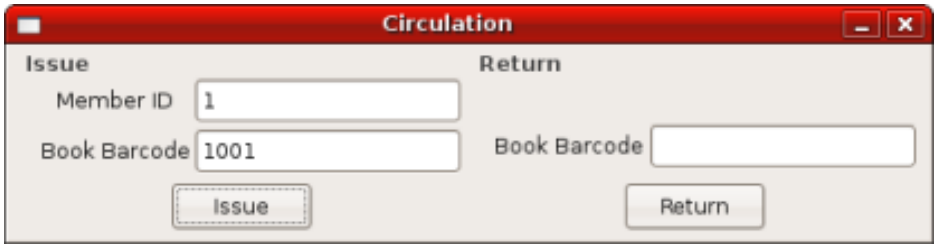


Dalla finestra Member, l'utente dovrebbe poter gestire i membri. Quindi dovrebbe essere possibile *aggiungere*, *modificare* e *eliminare* i membri come in figura sotto

Simile alla finestra dei membri, la finestra del catalogo permette all'utente di *aggiungere*, *modificare* e *eliminare* i libri:



La finestra dei movimenti dovrebbe gestire i prestiti e le restituzioni dei libri:



8.3 Panoramica del codice PyGTK

Come si può vedere dal codice, la maggior parte del codice è legato a PyGTK e la sua struttura è molto simile per le diverse finestre. Le finestre di questa applicazione sono disegnate utilizzando il costruttore di GUI Glade. Si dovrebbero assegnare nomi sensati ai widget che si andrà ad utilizzare nel codice. Nella finestra principale, tutte le voci del menu hanno nomi come *circulation*, *catalog*, *member*, *quit* e *about*.

La classe `gtk.glade.XML` è utilizzata analizzare il file Glade e quindi creare gli oggetti widget dell'interfaccia grafica. Ecco come analizzare e accedere agli oggetti:

```
import gtk.glade
xmlobj = gtk.glade.XML('/path/to/file.glade')
widget = xmlobj.get_widget('widget_name')
```

Nel file `mainwindow.py`, si può vedere il codice:

```
curdir = os.path.abspath(os.path.dirname(__file__))
xml = os.path.join(curdir, 'glade', 'mainwindow.glade')
xmlobj = gtk.glade.XML(xml)

self.mainwindow = xmlobj.get_widget('mainwindow')
```

Il nome del widget della finestra principale è `mainwindow`. In maniera simile, gli altri widget vengono recuperati così:

```
circulation = xmlobj.get_widget('circulation')
member = xmlobj.get_widget('member')
quit = xmlobj.get_widget('quit')
catalog = xmlobj.get_widget('catalog')
about = xmlobj.get_widget('about')
```

Poi questi widget vengono connessi a certi eventi:

```
self.mainwindow.connect('delete_event', self.delete_event)
quit.connect('activate', self.delete_event)
circulation.connect('activate', self.on_circulation_activate)
member.connect('activate', self.on_member_activate)
catalog.connect('activate', self.on_catalog_activate)
about.connect('activate', self.on_about_activate)
```

Il `delete_event` è l'evento scatenato durante la chiusura della finestra utilizzando l'apposito bottone. L'evento `activate` è lanciato quando il menu viene selezionato. I widget sono connessi a certe funzioni di callback per certi eventi.

Possiamo vedere dal codice sopra che la finestra principale è connessa al metodo `on_delete_event` per il `delete_event`. Il widget `quit` è anche connesso allo stesso metodo per l'evento `activate`:

```
def on_delete_event(self, *args):
    gtk.main_quit()
```

La funzione di callback chiama semplicemente la funzione `main_quit`.

8.4 Il codice

Ecco il file `zcalib.py`:

```
import registry
import mainwindow

if __name__ == '__main__':
    registry.initialize()
    try:
        mainwindow.main()
    except KeyboardInterrupt:
        import sys
        sys.exit(1)
```

Qui vengono importati due moduli: `registry` e `mainwindow`. Poi il registro viene analizzato e viene chiamata la funzione main di `mainwindow`. Se l'utente sta cercando di uscire dall'applicazione usando *Ctrl+C*, il sistema uscirà normalmente poiché abbiamo intercettato l'eccezione `KeyboardInterrupt`.

Questo è il modulo `registry.py`:

```
import sys
from zope.component import getGlobalSiteManager

from interfaces import IMember
from interfaces import IBook
from interfaces import ICirculation
from interfaces import IDbOperation


def initialize_rdb():
    from interfaces import IRelationalDatabase
    from relationaldatabase import RelationalDatabase
    from member import MemberRDbOperation
    from catalog import BookRDbOperation
    from circulation import CirculationRDbOperation

    gsm = getGlobalSiteManager()
    db = RelationalDatabase()
    gsm.registerUtility(db, IRelationalDatabase)

    gsm.registerAdapter(MemberRDbOperation,
                        (IMember,),
                        IDbOperation)

    gsm.registerAdapter(BookRDbOperation,
                        (IBook,),
                        IDbOperation)

    gsm.registerAdapter(CirculationRDbOperation,
                        (ICirculation,),
                        IDbOperation)

def initialize_odb():
    from interfaces import IObjectDatabase
    from objectdatabase import ObjectDatabase
    from member import MemberODbOperation
    from catalog import BookODbOperation
    from circulation import CirculationODbOperation

    gsm = getGlobalSiteManager()
    db = ObjectDatabase()
    gsm.registerUtility(db, IObjectDatabase)
```

```
    gsm.registerAdapter(MemberODbOperation,
                        (IMember,),
                        IDbOperation)

    gsm.registerAdapter(BookODbOperation,
                        (IBook,),
                        IDbOperation)

    gsm.registerAdapter(CirculationODbOperation,
                        (ICirculation,),
                        IDbOperation)

def check_use_relational_db():
    use_rdb = False
    try:
        arg = sys.argv[1]
        if arg == '-r':
            return True
    except IndexError:
        pass
    return use_rdb

def initialize():
    use_rdb = check_use_relational_db()
    if use_rdb:
        initialize_rdb()
    else:
        initialize_odb()
```

Diamo uno sguardo alla funzione `initialize` che stiamo chiamando dal modulo principale, `zcalib.py`. La funzione `initialize` per prima cosa controlla quale db è in uso, il database relazionale (RDB) o il database ad oggetti (ODB) e questo controllo è fatto nella funzione `check_use_relational_db`. Se è stata passata dalla linea di comando l'opzione -r, la funzione chiamerà `initialize_rdb` altrimenti `initialize_odb`. Se la funzione RDB viene chiamata, essa configurerà tutti i componenti legati a RDB altrimenti se viene chiamata la funzione ODB, verranno configurati tutti i componenti legati a ODB.

Ecco il file `mainwindow.py`:

```
import os
import gtk
import gtk.glade

from circulationwindow import circulationwindow
from catalogwindow import catalogwindow
from memberwindow import memberwindow

class MainWindow(object):

    def __init__(self):
        curdir = os.path.abspath(os.path.dirname(__file__))
        xml = os.path.join(curdir, 'glade', 'mainwindow.glade')
        xmlobj = gtk.glade.XML(xml)

        self.mainwindow = xmlobj.get_widget('mainwindow')
        circulation = xmlobj.get_widget('circulation')
        member = xmlobj.get_widget('member')
        quit = xmlobj.get_widget('quit')
        catalog = xmlobj.get_widget('catalog')
```

```
        about = xmlobj.get_widget('about')

        self.mainwindow.connect('delete_event', self.delete_event)
        quit.connect('activate', self.delete_event)

        circulation.connect('activate', self.on_circulation_activate)
        member.connect('activate', self.on_member_activate)
        catalog.connect('activate', self.on_catalog_activate)
        about.connect('activate', self.on_about_activate)

    def delete_event(self, *args):
        gtk.main_quit()

    def on_circulation_activate(self, *args):
        circulationwindow.show_all()

    def on_member_activate(self, *args):
        memberwindow.show_all()

    def on_catalog_activate(self, *args):
        catalogwindow.show_all()

    def on_about_activate(self, *args):
        pass

    def run(self):
        self.mainwindow.show_all()

def main():
    mainwindow = MainWindow()
    mainwindow.run()
    gtk.main()
```

La funzione `main` crea un'istanza della classe `MainWindow` che inizializza tutti i widget.

Ecco qui *memberwindow.py*:

```
import os
import gtk
import gtk.glade

from zope.component import getAdapter

from components import Member
from interfaces import IDbOperation


class MemberWindow(object):

    def __init__(self):
        curdir = os.path.abspath(os.path.dirname(__file__))
        xml = os.path.join(curdir, 'glade', 'memberwindow.glade')
        xmlobj = gtk.glade.XML(xml)

        self.memberwindow = xmlobj.get_widget('memberwindow')
        self.number = xmlobj.get_widget('number')
        self.name = xmlobj.get_widget('name')
        add = xmlobj.get_widget('add')
        update = xmlobj.get_widget('update')
```

```
        delete = xmlobj.get_widget('delete')
        close = xmlobj.get_widget('close')
        self.treeview = xmlobj.get_widget('treeview')

        self.memberwindow.connect('delete_event', self.on_delete_event)
        add.connect('clicked', self.on_add_clicked)
        update.connect('clicked', self.on_update_clicked)
        delete.connect('clicked', self.on_delete_clicked)
        close.connect('clicked', self.on_delete_event)

        self.initialize_list()

    def show_all(self):
        self.populate_list_store()
        self.memberwindow.show_all()

    def populate_list_store(self):
        self.list_store.clear()
        member = Member()
        memberdboperation = getAdapter(member, IDbOperation)
        members = memberdboperation.get()
        for member in members:
            number = member.number
            name = member.name
            self.list_store.append((member, number, name,))

    def on_delete_event(self, *args):
        self.memberwindow.hide()
        return True

    def initialize_list(self):
        self.list_store = gtk.ListStore(object, str, str)
        self.treeview.set_model(self.list_store)
        tvcolumn = gtk.TreeViewColumn('Member Number')
        self.treeview.append_column(tvcolumn)

        cell = gtk.CellRendererText()
        tvcolumn.pack_start(cell, True)
        tvcolumn.add_attribute(cell, 'text', 1)

        tvcolumn = gtk.TreeViewColumn('Member Name')
        self.treeview.append_column(tvcolumn)

        cell = gtk.CellRendererText()
        tvcolumn.pack_start(cell, True)
        tvcolumn.add_attribute(cell, 'text', 2)

    def on_add_clicked(self, *args):
        number = self.number.get_text()
        name = self.name.get_text()
        member = Member()
        member.number = number
        member.name = name
        self.add(member)
        self.list_store.append((member, number, name,))

    def add(self, member):
        memberdboperation = getAdapter(member, IDbOperation)
```

```
        memberdboperation.add()

    def on_update_clicked(self, *args):
        number = self.number.get_text()
        name = self.name.get_text()
        treeselection = self.treeview.get_selection()
        model, iter = treeselection.get_selected()
        if not iter:
            return
        member = self.list_store.get_value(iter, 0)
        member.number = number
        member.name = name
        self.update(member)
        self.list_store.set(iter, 1, number, 2, name)

    def update(self, member):
        memberdboperation = getAdapter(member, IDbOperation)
        memberdboperation.update()

    def on_delete_clicked(self, *args):
        treeselection = self.treeview.get_selection()
        model, iter = treeselection.get_selected()
        if not iter:
            return
        member = self.list_store.get_value(iter, 0)
        self.delete(member)
        self.list_store.remove(iter)

    def delete(self, member):
        memberdboperation = getAdapter(member, IDbOperation)
        memberdboperation.delete()

memberwindow = MemberWindow()
```

Ecco qui *components.py*:

```
from zope.interface import implements

from interfaces import IBook
from interfaces import IMember
from interfaces import ICirculation

class Book(object):

    implements(IBook)

    barcode = ""
    title = ""
    author = ""

class Member(object):

    implements(IMember)

    number = ""
    name = ""

class Circulation(object):
```

```
    implements(ICirculation)

    book = Book()
    member = Member()
```

Ecco qui *interfaces.py*:

```
from zope.interface import Interface
from zope.interface import Attribute


class IBook(Interface):

    barcode = Attribute("Barcode")
    author = Attribute("Author of book")
    title = Attribute("Title of book")


class IMember(Interface):

    number = Attribute("ID number")
    name = Attribute("Name of member")


class ICirculation(Interface):

    book = Attribute("A book")
    member = Attribute("A member")


class IRelationalDatabase(Interface):

    def commit():
        pass

    def rollback():
        pass

    def cursor():
        pass

    def get_next_id():
        pass


class IObjectDatabase(Interface):

    def commit():
        pass

    def rollback():
        pass

    def container():
        pass

    def get_next_id():
        pass
```

```
class IDbOperation(Interface):

    def get():
        pass

    def add():
        pass

    def update():
        pass

    def delete():
        pass
```

Ecco qui *member.py*:

```
from zope.interface import implements
from zope.component import getUtility
from zope.component import adapts

from components import Member

from interfaces import IRelationalDatabase
from interfaces import IObjectDatabase
from interfaces import IMember
from interfaces import IDbOperation


class MemberRDbOperation(object):

    implements(IDbOperation)
    adapts(IMember)

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

    def get(self):
        db = getUtility(IRelationalDatabase)
        cr = db.cursor()
        number = self.member.number
        if number:
            cr.execute("""SELECT
                            id,
                            number,
                            name
                          FROM members
                          WHERE number = ?""",
                       (number,))
        else:
            cr.execute("""SELECT
                            id,
                            number,
                            name
                          FROM members""")
        rst = cr.fetchall()
        cr.close()
        members = []
```

```
        for record in rst:
            id = record['id']
            number = record['number']
            name = record['name']
            member = Member()
            member.id = id
            member.number = number
            member.name = name
            members.append(member)
        return members

    def add(self):
        db = getUtility(IRelationalDatabase)
        cr = db.cursor()
        next_id = db.get_next_id("members")
        number = self.member.number
        name = self.member.name
        cr.execute("""INSERT INTO members
                        (id, number, name)
                      VALUES (?, ?, ?)""",
                   (next_id, number, name))
        cr.close()
        db.commit()
        self.member.id = next_id

    def update(self):
        db = getUtility(IRelationalDatabase)
        cr = db.cursor()
        number = self.member.number
        name = self.member.name
        id = self.member.id
        cr.execute("""UPDATE members
                        SET
                           number = ?,
                           name = ?
                      WHERE id = ?""",
                   (number, name, id))
        cr.close()
        db.commit()

    def delete(self):
        db = getUtility(IRelationalDatabase)
        cr = db.cursor()
        id = self.member.id
        cr.execute("""DELETE FROM members
                      WHERE id = ?""",
                   (id,))
        cr.close()
        db.commit()


class MemberODbOperation(object):

    implements(IDbOperation)
    adapts(IMember)

    def __init__(self, member):
        self.member = member
```

```
    def get(self):
        db = getUtility(IObjectDatabase)
        zcalibdb = db.container()
        members = zcalibdb['members']
        return members.values()

    def add(self):
        db = getUtility(IObjectDatabase)
        zcalibdb = db.container()
        members = zcalibdb['members']
        number = self.member.number
        if number in [x.number for x in members.values()]:
            db.rollback()
            raise Exception("Duplicate key")
        next_id = db.get_next_id('members')
        self.member.id = next_id
        members[next_id] = self.member
        db.commit()

    def update(self):
        db = getUtility(IObjectDatabase)
        zcalibdb = db.container()
        members = zcalibdb['members']
        id = self.member.id
        members[id] = self.member
        db.commit()

    def delete(self):
        db = getUtility(IObjectDatabase)
        zcalibdb = db.container()
        members = zcalibdb['members']
        id = self.member.id
        del members[id]
        db.commit()
```

8.5 PySQLite

8.6 ZODB

8.7 Conclusions

CHAPTER 9

Riferimenti

9.1 adaptedBy

Questa funzione permette di trovare le interfacce adattate.

- Posizione: `zope.component`
- Firma: *adaptedBy(object)*

Esempio:

```
>>> from zope.interface import implements
>>> from zope.component import adapts
>>> from zope.component import adaptedBy

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest

>>> adaptedBy(FrontDeskNG)
(<InterfaceClass __builtin__.IGuest>,)
```

9.2 adapter

Qualsiasi tipo di oggetto può essere un adattatore, è possibile utilizzare il decoratore *adapter* per dichiarare che un oggetto chiamabile adatta qualche interfaccia (o classe)

- Posizione: `zope.component`
- Firma: *adapter(*interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implementer
>>> from zope.component import adapter
>>> from zope.interface import implements
```

```
>>> class IJob(Interface):
...     """A job"""

>>> class Job(object):
...     implements(IJob)

>>> class IPerson(Interface):
...
...     name = Attribute("Name")
...     job = Attribute("Job")

>>> class Person(object):
...     implements(IPerson)
...
...     name = None
...     job = None

>>> @implementer(IJob)
... @adapter(IPerson)
... def personJob(person):
...     return person.job

>>> jack = Person()
>>> jack.name = "Jack"
>>> jack.job = Job()
>>> personJob(jack)
<Job object at ...>
```

9.3 adapts

Questa funzione permette di dichiarare le interfacce adattate dall'adapter

- Posizione: `zope.component`
- Firma: *adapts(*interfaces)*

Esempio:

```
>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }
```

9.4 alsoProvides

Dichiara le interfacce fornite direttamente da un oggetto. Gli argomenti dopo l'oggetto sono una o più interfacce. Le interfacce fornite vengono aggiunte alle interfacce già dichiarate per l'oggetto.

- Posizione: `zope.interface`
- Firma: *alsoProvides(object, *interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.interface import alsoProvides

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class Person(object):
...
...     implements(IDesk)
...     name = u""

>>> jack = Person()
>>> jack.name = "Jack"
>>> jack.college = "New College"
>>> alsoProvides(jack, IStudent)

Si può testare così:

>>> from zope.interface import providedBy
>>> IStudent in providedBy(jack)
True
```

9.5 Attribute

Con questa classe è possibile definire i normali attributi di una interfaccia.

- Posizione: `zope.interface`
- Firma: *Attribute(name, doc='')*
- Vedi anche: *Interface*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IPerson(Interface):
...
```

```
...     name = Attribute("Name of person")
...     email = Attribute("Email Address")
```

9.6 classImplements

Dichiara le interfacce aggiuntive implementate dalle istanze di una classe. Gli argomenti dopo la classe sono una o più interfacce. Le interfacce fornite vengono aggiunte alle interfacce già dichiarate.

- Posizione: `zope.interface`
- Firma: *classImplements(cls, *interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.interface import classImplements

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class Person(object):
...
...     implements(IDesk)
...     name = u""
...     college = u""

>>> classImplements(Person, IStudent)
>>> jack = Person()
>>> jack.name = "Jack"
>>> jack.college = "New College"

Si può testare così:

>>> from zope.interface import providedBy
>>> IStudent in providedBy(jack)
True
```

9.7 classImplementsOnly

Dichiara le sole interfacce implementate dalle istanze di una classe. Gli argomenti dopo la classe sono una o più interfacce. Le interfacce fornite rimpiazzano le dichiarazioni precedenti.

- Posizione: `zope.interface`
- Firma: *classImplementsOnly(cls, *interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.interface import classImplementsOnly

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class Person(object):
...
...     implements(IPerson)
...     college = u""

>>> classImplementsOnly(Person, IStudent)
>>> jack = Person()
>>> jack.college = "New College"

Si può testare così:

>>> from zope.interface import providedBy
>>> IPerson in providedBy(jack)
False
>>> IStudent in providedBy(jack)
True
```

9.8 classProvides

Normalmente se una classe implementa una particolare interfaccia, l'istanza di questa classe fornirà l'interfaccia implementata da questa classe. Se però si vuole che la classe stessa fornisca un'interfaccia, si può utilizzare questa funzione.

- Posizione: `zope.interface`
- Firma: *classProvides(*interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import classProvides

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class Person(object):
...
...     classProvides(IPerson)
...     name = u"Jack"

Si può testare così:
```

```
>>> from zope.interface import providedBy
>>> IPerson in providedBy(Person)
True
```

9.9 ComponentLookupError

Questa è l'eccezione che viene sollevata quando una ricerca di un componente fallisce.

Esempio:

```
>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> person = object()
>>> getAdapter(person, IPerson, 'not-exists')
Traceback (most recent call last):
...
ComponentLookupError: ...
```

9.10 createObject

Crea un oggetto usando una factory.

Cerca la named factory nel sito corrente e la chiama con i parametri forniti. Se non può essere trovata alcuna factory, viene sollevata l'eccezione `ComponentLookupError` altrimenti restituisce l'oggetto creato.

Può essere fornito come argomento keyword un context per forzare la ricerca della factory in una posizione diversa dal sito corrente. Ovviamente questo significa che è impossibile passare un argomento keyword alla factory chiamato "context".

- Posizione: `zope.component`
- Firma: *createObject(factory_name, *args, **kwargs)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IDatabase(Interface):
...
...     def getConnection():
...         """Return connection object"""

>>> class FakeDb(object):
...
...     implements(IDatabase)
...
...     def getConnection(self):
...         return "connection"

>>> from zope.component.factory import Factory

>>> factory = Factory(FakeDb, 'FakeDb')
```

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(factory, IFactory, 'fakedb')

>>> from zope.component import createObject
>>> createObject('fakedb')
<FakeDb object at ...>
```

9.11 Declaration

Non deve essere usata direttamente.

9.12 directlyProvidedBy

Questa funzione restituirà le interfacce fornite direttamente dall'oggetto passato come argomento.

- Posizione: `zope.interface`
- Firma: *directlyProvidedBy(object)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class ISmartPerson(Interface):
...     pass

>>> class Person(object):
...
...     implements(IPerson)
...     name = u""

>>> jack = Person()
>>> jack.name = u"Jack"
>>> jack.college = "New College"
>>> alsoProvides(jack, ISmartPerson, IStudent)

>>> from zope.interface import directlyProvidedBy

>>> jack_dp = directlyProvidedBy(jack)
>>> IPerson in jack_dp.interfaces()
False
>>> IStudent in jack_dp.interfaces()
```

```
True
>>> ISmartPerson in jack_dp.interfaces()
True
```

9.13 directlyProvides

Dichiara le interfacce fornite direttamente da un oggetto. Gli argomenti dopo l'oggetto sono una o più interfacce. Le interfacce fornite rimpiazzano le interfacce già dichiarate in precedenza dall'oggetto.

- Posizione: `zope.interface`
- Firma: *directlyProvides(object, *interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class ISmartPerson(Interface):
...     pass

>>> class Person(object):
...
...     implements(IPerson)
...     name = u""

>>> jack = Person()
>>> jack.name = u"Jack"
>>> jack.college = "New College"
>>> alsoProvides(jack, ISmartPerson, IStudent)

>>> from zope.interface import directlyProvidedBy

>>> jack_dp = directlyProvidedBy(jack)
>>> ISmartPerson in jack_dp.interfaces()
True
>>> IPerson in jack_dp.interfaces()
False
>>> IStudent in jack_dp.interfaces()
True
>>> from zope.interface import providedBy

>>> ISmartPerson in providedBy(jack)
True

>>> from zope.interface import directlyProvides
>>> directlyProvides(jack, IStudent)

>>> jack_dp = directlyProvidedBy(jack)
```

```
>>> ISmartPerson in jack_dp.interfaces()
False
>>> IPerson in jack_dp.interfaces()
False
>>> IStudent in jack_dp.interfaces()
True

>>> ISmartPerson in providedBy(jack)
False
```

9.14 getAdapter

Recupera un adapter per un oggetto verso una specifica interfaccia. Restituisce un adapter che può adattare l'oggetto all'interfaccia. Se non può essere trovato alcun adapter, solleva `ComponentLookupError`.

- Posizione: `zope.interface`
- Firma: *getAdapter(object, interface=Interface, name=u'', context=None)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place
```

```
>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(FrontDeskNG,
...                     (IGuest,), IDesk, 'ng')

>>> getAdapter(jack, IDesk, 'ng')
<FrontDeskNG object at ...>
```

9.15 getAdapterInContext

Al posto di questa funzione, utilizzare l'argomento *context* della funzione *getAdapter*.

- Posizione: `zope.component`
- Firma: *getAdapterInContext(object, interface, context)*
- Vedi anche: *queryAdapterInContext*

Esempio:

```
>>> from zope.component.globalregistry import BaseGlobalComponents
>>> from zope.component import IComponentLookup
>>> sm = BaseGlobalComponents()

>>> class Context(object):
...     def __init__(self, sm):
...         self.sm = sm
...     def __conform__(self, interface):
...         if interface.isOrExtends(IComponentLookup):
...             return self.sm

>>> context = Context(sm)

>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
```

```
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> sm.registerAdapter(FrontDeskNG,
...                    (IGuest,), IDesk)

>>> from zope.component import getAdapterInContext

>>> getAdapterInContext(jack, IDesk, sm)
<FrontDeskNG object at ...>
```

9.16 getAdapters

Cerca tutti gli adapter corrispondenti per degli oggetti e per una interfaccia fornita. Restituisce una lista di adapter che corrispondono. Se un adapter ha un nome, viene restituito solo l'adapter più specifico.

- Posizione: `zope.component`
- Firma: *getAdapters(objects, provided, context=None)*

Esempio:

```
>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
```

```
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerAdapter(FrontDeskNG, name='ng')

>>> from zope.component import getAdapters
>>> list(getAdapters((jack,), IDesk))
[(u'ng', <FrontDeskNG object at ...>)]
```

9.17 getAllUtilitiesRegisteredFor

Restituisce tutte le utility registrate per un'interfaccia. Questo include anche le utility sovrascritte. Il valore di ritorno è un iterabile di istanze di utility.

- Posizione: `zope.component`
- Firma: *getAllUtilitiesRegisteredFor(interface)*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         print "Hello", name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter)

>>> from zope.component import getAllUtilitiesRegisteredFor

>>> getAllUtilitiesRegisteredFor(IGreeter)
[<Greeter object at ...>]
```

9.18 getFactoriesFor

Restituisce una tupla (nome, factory) delle factory registrate che creano oggetti che implementano l'interfaccia fornita.

- Posizione: `zope.component`
- Firma: *getFactoriesFor(interface, context=None)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IDatabase(Interface):
...
...     def getConnection():
...         """Return connection object"""

>>> class FakeDb(object):
...
...     implements(IDatabase)
...
...     def getConnection(self):
...         return "connection"

>>> from zope.component.factory import Factory

>>> factory = Factory(FakeDb, 'FakeDb')

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(factory, IFactory, 'fakedb')

>>> from zope.component import getFactoriesFor

>>> list(getFactoriesFor(IDatabase))
[(u'fakedb', <Factory for <class 'FakeDb'>>)]
```

9.19 getFactoryInterfaces

Trova le interfacce implementate da una factory. Trova la factory più vicina al contesto con il nome specificato e restituisce l'interfaccia o la tupla dell'interfaccia che gli oggetti istanza creati forniranno.

- Posizione: `zope.component`
- Firma: *getFactoryInterfaces(name, context=None)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IDatabase(Interface):
...
```

```
...     def getConnection():
...         """Return connection object"""

>>> class FakeDb(object):
...
...     implements(IDatabase)
...
...     def getConnection(self):
...         return "connection"

>>> from zope.component.factory import Factory

>>> factory = Factory(FakeDb, 'FakeDb')

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(factory, IFactory, 'fakedb')

>>> from zope.component import getFactoryInterfaces

>>> getFactoryInterfaces('fakedb')
<implementedBy __builtin__.FakeDb>
```

9.20 getGlobalSiteManager

Restituisce il global site manager. Questa funzione non dovrebbe mai fallire e dovrebbe sempre restituire un oggetto che fornisce *IGlobalSiteManager*

- Posizione: `zope.component`
- Firma: *getGlobalSiteManager()*

Esempio:

```
>>> from zope.component import getGlobalSiteManager
>>> from zope.component import globalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm is globalSiteManager
True
```

9.21 getMultiAdapter

Cerca e restituisce un multi-adapter che può adattare degli oggetti ad una certa interfaccia. Se non può essere trovato alcun adapter, solleva `ComponentLookupError`. La stringa vuota come nome è riservata per gli adapter senza nome. I metodi per gli adapter senza nome spesso chiamano i metodi per i named adapter con una stringa vuota come nome.

- Posizione: `zope.component`
- Firma: *getMultiAdapter(objects, interface=Interface, name='', context=None)*
- Vedi anche: *queryMultiAdapter*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class IAdapteeOne(Interface):
...     pass

>>> class IAdapteeTwo(Interface):
...     pass

>>> class IFunctionality(Interface):
...     pass

>>> class MyFunctionality(object):
...     implements(IFunctionality)
...     adapts(IAdapteeOne, IAdapteeTwo)
...
...     def __init__(self, one, two):
...         self.one = one
...         self.two = two

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerAdapter(MyFunctionality)

>>> class One(object):
...     implements(IAdapteeOne)

>>> class Two(object):
...     implements(IAdapteeTwo)

>>> one = One()
>>> two = Two()

>>> from zope.component import getMultiAdapter

>>> getMultiAdapter((one,two), IFunctionality)
<MyFunctionality object at ...>

>>> myfunctionality = getMultiAdapter((one,two), IFunctionality)
>>> myfunctionality.one
<One object at ...>
>>> myfunctionality.two
<Two object at ...>
```

9.22 getSiteManager

Prende il site manager più vicino al contesto dato. Se il *context* è *None*, restituisce il global site manager. Se il *context* non è *None*, ci si aspetta di poter trovare un adapter dal *context* a *IComponentLookup*. Se non viene trovato alcun adapter, viene sollevato *ComponentLookupError*.

- Posizione: `zope.component`
- Firma: *getSiteManager(context=None)*

Esempio 1:

```
>>> from zope.component.globalregistry import BaseGlobalComponents
>>> from zope.component import IComponentLookup
>>> sm = BaseGlobalComponents()

>>> class Context(object):
...     def __init__(self, sm):
...         self.sm = sm
...     def __conform__(self, interface):
...         if interface.isOrExtends(IComponentLookup):
...             return self.sm

>>> context = Context(sm)

>>> from zope.component import getSiteManager

>>> lsm = getSiteManager(context)
>>> lsm is sm
True
```

Esempio 2:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> sm = getSiteManager()
>>> gsm is sm
True
```

9.23 getUtilitiesFor

Ricerca le utility registrate che forniscono un'interfaccia. Restituisce un iterabile delle coppie nome-utility.

- Posizione: `zope.component`
- Firma: *getUtilitiesFor(interface)*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         print "Hello", name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter)
```

```
>>> from zope.component import getUtilitiesFor

>>> list(getUtilitiesFor(IGreeter))
[(u'', <Greeter object at ...>)]
```

9.24 getUtility

Recupera l'utility che fornisce l'interfaccia. Restituisce l'utility più vicina al contesto e che implementa una una specifica interfaccia. Se non nè vengono trovate, viene sollevata `ComponentLookupError`.

- Posizione: `zope.component`
- Firma: *getUtility(interface, name='', context=None)*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         return "Hello " + name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter)

>>> from zope.component import getUtility

>>> getUtility(IGreeter).greet('Jack')
'Hello Jack'
```

9.25 handle

Chiama tutti gli handler per gli oggetti dati. Gli handler sono fabbriche di subscription adapter che non restituiscono nulla. Essi fanno tutto il loro lavoro quando vengono chiamati. Gli handler sono tipicamente utilizzati per gestire gli eventi.

- Posizione: `zope.component`
- Firma: *handle(*objects)*

Esempio:

```
>>> import datetime
```

```
>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()

>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IDocumentCreated(Interface):
...     doc = Attribute("The document that was created")

>>> class DocumentCreated(object):
...     implements(IDocumentCreated)
...
...     def __init__(self, doc):
...         self.doc = doc


>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()

>>> from zope.component import adapter

>>> @adapter(IDocumentCreated)
... def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()


>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerHandler(documentCreated)

>>> from zope.component import handle

>>> handle(DocumentCreated(doc))
>>> doc.created.__class__.__name__
'datetime'
```

9.26 implementedBy

Restituisce le interfacce implementate dalle istanze di una certa classe.

- Posizione: `zope.interface`
- Firma: *implementedBy(class_)*

Esempio 1:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
```

```
...     implements(IGreeter)
...
...     def greet(self, name):
...         print "Hello", name

>>> from zope.interface import implementedBy
>>> implementedBy(Greeter)
<implementedBy __builtin__.Greeter>
```

Esempio 2:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IPerson(Interface):
...     name = Attribute("Name of person")

>>> class ISpecial(Interface):
...     pass

>>> class Person(object):
...     implements(IPerson)
...     name = u""

>>> from zope.interface import classImplements
>>> classImplements(Person, ISpecial)

>>> from zope.interface import implementedBy

To get a list of all interfaces implemented by that class::

>>> [x.__name__ for x in implementedBy(Person)]
['IPerson', 'ISpecial']
```

9.27 implementer

Crea un decoratore per dichiarare le interfacce implementate da una factory. Viene restituito un oggetto chiamabile che fa una dichiarazione di implementazione sugli oggetti che gli vengono passati.

- Posizione: `zope.interface`
- Firma: *implementer(*interfaces)*

Esempio:

```
>>> from zope.interface import implementer
>>> class IFoo(Interface):
...     pass
>>> class Foo(object):
...     implements(IFoo)

>>> @implementer(IFoo)
... def foocreator():
...     foo = Foo()
...     return foo
```

```
>>> list(implementedBy(foocreator))
[<InterfaceClass __builtin__.IFoo>]
```

9.28 implements

Dichiara le interfacce implementate dalle istanze di una classe. Questa funzione è chiamata all'interno di una definizione di una classe. Gli argomenti sono una o più interfacce. Le interfacce fornite sono aggiunte a quelle già dichiarate in precedenza. Le dichiarazioni precedenti, incluse le dichiarazioni delle classi base, vengono preservate, a meno che non sia stata utilizzata `implementsOnly`.

- Posizione: `zope.interface`
- Firma: *implements(*interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class Person(object):
...
...     implements(IPerson)
...     name = u""

>>> jack = Person()
>>> jack.name = "Jack"

Si può testare così:

>>> from zope.interface import providedBy
>>> IPerson in providedBy(jack)
True
```

9.29 implementsOnly

Dichiara le sole interfacce implementate dalle istanze di una classe. Questa funzione è chiamata all'interno di una definizione di classe. Gli argomenti sono una o più interfacce. Le dichiarazioni precedenti, incluse le dichiarazioni delle classi base, vengono sovrascritte.

- Posizione: `zope.interface`
- Firma: *implementsOnly(*interfaces)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.interface import implementsOnly
```

```
>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class Person(object):
...
...     implements(IPerson)
...     name = u""

>>> class NewPerson(Person):
...     implementsOnly(IStudent)
...     college = u""

>>> jack = NewPerson()
>>> jack.college = "New College"

Si può testare così:

>>> from zope.interface import providedBy
>>> IPerson in providedBy(jack)
False
>>> IStudent in providedBy(jack)
True
```

9.30 Interface

Con questa classe si possono definire le interfacce. Per definire un'interfaccia, basta ereditare dalla classe `Interface`.

- Posizione: `zope.interface`
- Firma: *Interface(name, doc='')*

Esempio 1:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")
...     email = Attribute("Email Address")
```

Esempio 2:

```
>>> from zope.interface import Interface

>>> class IHost(Interface):
...
...     def goodmorning(guest):
...         """Say good morning to guest"""
```

9.31 moduleProvides

Dichiara le interfacce fornite da un modulo. Questa funzione è utilizzata nella definizione di un modulo. Gli argomenti sono una o più interfacce. Le interfacce fornite vengono utilizzate per creare la definizione di interfaccia degli oggetti diretti del modulo. Verrà sollevato un errore se il modulo ha già una dichiarazione di interfaccia. In altre parole, è un errore chiamare questa funzione più di una volta nella definizione di un modulo.

Questa funzione è fornita per comodità. Essa fornisce un modo più conveniente per chiamare `directlyProvides` su un modulo.

- Posizione: `zope.interface`
- Firma: *moduleProvides(*interfaces)*
- Vedi anche: *directlyProvides*

You can see an example usage in *zope.component* source itself. The *__init__.py* file has a statement like this:

```
moduleProvides(IComponentArchitecture,
               IComponentRegistrationConvenience)
```

So, the *zope.component* provides two interfaces: *IComponentArchitecture* and *IComponentRegistrationConvenience*.

9.32 noLongerProvides

Rimuove un'interfaccia dalla lista delle interfacce fornite direttamente da un oggetto.

- Posizione: `zope.interface`
- Firma: *noLongerProvides(object, interface)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.interface import classImplements

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class IStudent(Interface):
...
...     college = Attribute("Name of college")

>>> class Person(object):
...
...     implements(IPerson)
...     name = u""

>>> jack = Person()
>>> jack.name = "Jack"
>>> jack.college = "New College"
>>> directlyProvides(jack, IStudent)

Si può testare così:

>>> from zope.interface import providedBy
```

```
>>> IPerson in providedBy(jack)
True
>>> IStudent in providedBy(jack)
True
>>> from zope.interface import noLongerProvides
>>> noLongerProvides(jack, IStudent)
>>> IPerson in providedBy(jack)
True
>>> IStudent in providedBy(jack)
False
```

9.33 provideAdapter

Si raccomanda di utilizzare *registerAdapter* al posto di questa funzione.

9.34 provideHandler

Si raccomanda di utilizzare *registerHandler* al posto di questa funzione.

9.35 provideSubscriptionAdapter

Si raccomanda di utilizzare *registerSubscriptionAdapter* al posto di questa funzione.

9.36 provideUtility

Si raccomanda di utilizzare *registerUtility* al posto di questa funzione.

9.37 providedBy

Verifica se l'interfaccia è fornita dall'oggetto dato. Restituisce true se l'oggetto dichiara di fornire l'interfaccia, anche se dichiara di fornire un'interfaccia che estende l'interfaccia data.

- Posizione: `zope.interface`
- Firma: *providedBy(object)*

Esempio 1:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IPerson(Interface):
...
...     name = Attribute("Name of person")

>>> class Person(object):
...
```

```
...     implements(IPerson)
...     name = u""

>>> jack = Person()
>>> jack.name = "Jack"

Si può testare così:

>>> from zope.interface import providedBy
>>> IPerson in providedBy(jack)
True
```

Esempio 2:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IPerson(Interface):
...     name = Attribute("Name of person")

>>> class ISpecial(Interface):
...     pass

>>> class Person(object):
...     implements(IPerson)
...     name = u""

>>> from zope.interface import classImplements
>>> classImplements(Person, ISpecial)
>>> from zope.interface import providedBy
>>> jack = Person()
>>> jack.name = "Jack"

Ecco come vere la lista di tutte le interfacce fornite da questo oggetto::

>>> [x.__name__ for x in providedBy(jack)]
['IPerson', 'ISpecial']
```

9.38 queryAdapter

Cerca e restituisce un named adapter che può adattare un oggetto ad un'interfaccia. Se non può essere trovato alcun adapter restituisce il default.

- Posizione: `zope.component`
- Firma: *queryAdapter(object, interface=Interface, name=u'', default=None, context=None)*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
```

```
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(FrontDeskNG,
...                     (IGuest,), IDesk, 'ng')

>>> queryAdapter(jack, IDesk, 'ng')
<FrontDeskNG object at ...>
```

9.39 queryAdapterInContext

Cerca uno speciale adapter per adattare un oggetto a un'interfaccia.

Nota: Questo metodo dovrebbe essere utilizzato solo se è necessario fornire un context personalizzato per fornire una ricerca personalizzata. Altrimenti, chiamare l'interfaccia come in:

```
interface(object, default)
```

Restituisce un adapter che può adattare un oggetto a un'interfaccia. Se non può essere trovato alcun adapter, restituisce il default.

Il context viene adattato a IServiceService, e viene utilizzato il servizio 'Adapters' di questo adapter.

Se l'oggetto ha un metodo __conform__, questo metodo viene chiamato con l'interfaccia richiesta. Se il metodo restituisce un valore diverso da `None`, questo valore viene restituito. Altrimenti, se l'oggetto implementa già l'interfaccia, viene restituito l'oggetto.

- Posizione: `zope.component`
- Firma: *queryAdapterInContext(object, interface, context, default=None)*
- Vedi anche: *getAdapterInContext*

Esempio:

```
>>> from zope.component.globalregistry import BaseGlobalComponents
>>> from zope.component import IComponentLookup
>>> sm = BaseGlobalComponents()

>>> class Context(object):
...     def __init__(self, sm):
...         self.sm = sm
...     def __conform__(self, interface):
...         if interface.isOrExtends(IComponentLookup):
...             return self.sm

>>> context = Context(sm)

>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
```

```
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> sm.registerAdapter(FrontDeskNG,
...                    (IGuest,), IDesk)

>>> from zope.component import queryAdapterInContext

>>> queryAdapterInContext(jack, IDesk, sm)
<FrontDeskNG object at ...>
```

9.40 queryMultiAdapter

Cerca e restituisce un multi-adapter per adattare degli oggetti a un'interfaccia. Se non può essere trovato alcun adapter, restituisce il default. Il nome costituito dalla stringa vuota è riservato per gli adapters senza nome. I metodi per gli unnamed adapter spesso chiamano i metodi per i named adapter con una stringa vuota come nome.

- Posizione: `zope.component`
- Firma: *queryMultiAdapter(objects, interface=Interface, name=u'', default=None, context=None)*
- Vedi anche: *getMultiAdapter*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class IAdapteeOne(Interface):
...     pass

>>> class IAdapteeTwo(Interface):
...     pass

>>> class IFunctionality(Interface):
...     pass

>>> class MyFunctionality(object):
...     implements(IFunctionality)
...     adapts(IAdapteeOne, IAdapteeTwo)
...
...     def __init__(self, one, two):
...         self.one = one
...         self.two = two

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerAdapter(MyFunctionality)
```

```
>>> class One(object):
...     implements(IAdapteeOne)

>>> class Two(object):
...     implements(IAdapteeTwo)

>>> one = One()
>>> two = Two()

>>> from zope.component import queryMultiAdapter

>>> getMultiAdapter((one,two), IFunctionality)
<MyFunctionality object at ...>

>>> myfunctionality = queryMultiAdapter((one,two), IFunctionality)
>>> myfunctionality.one
<One object at ...>
>>> myfunctionality.two
<Two object at ...>
```

9.41 queryUtility

Questa funzione è utilizzata per cercare una utility che fornisce una certa interfaccia. Se non trova alcuna utility, restituisce il default.

- Posizione: `zope.component`
- Firma: *queryUtility(interface, name=''', default=None)*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         return "Hello " + name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter)

>>> from zope.component import queryUtility

>>> queryUtility(IGreeter).greet('Jack')
'Hello Jack'
```

9.42 registerAdapter

Questa funzione registra una factory di adapter.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registerAdapter(factory, required=None, provided=None, name=u'', info=u'')*
- Vedi anche: *unregisterAdapter*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(FrontDeskNG,
...                     (IGuest,), IDesk, 'ng')
```

```
Si può testare così:

>>> queryAdapter(jack, IDesk, 'ng')
<FrontDeskNG object at ...>
```

9.43 registeredAdapters

Restituisce un iterabile di *IAdapterRegistrations*. Queste registrazioni descrivono le attuali registrazioni degli adapter per l'oggetto.

- Posizione: `zope.component – IComponentRegistry`
- Firma: *registeredAdapters()*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)
```

```
>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(FrontDeskNG,
...                     (IGuest,), IDesk, 'ng2')


>>> reg_adapter = list(gsm.registeredAdapters())
>>> 'ng2' in [x.name for x in reg_adapter]
True
```

9.44 registeredHandlers

Restituisce un iterabile di *IHandlerRegistrations*. Queste registrazioni descrivono le attuali registrazioni degli handler per l'oggetto.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registeredHandlers()*

Esempio:

```
>>> import datetime

>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()

>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IDocumentCreated(Interface):
...     doc = Attribute("The document that was created")

>>> class DocumentCreated(object):
...     implements(IDocumentCreated)
...
...     def __init__(self, doc):
...         self.doc = doc


>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()

>>> from zope.component import adapter

>>> @adapter(IDocumentCreated)
... def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()


>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerHandler(documentCreated, info='ng3')
```

```
>>> reg_adapter = list(gsm.registeredHandlers())
>>> 'ng3' in [x.info for x in reg_adapter]
True

>>> gsm.registerHandler(documentCreated, name='ng4')
Traceback (most recent call last):
...
TypeError: Named handlers are not yet supported
```

9.45 registeredSubscriptionAdapters

Restituisce un iterabile di *ISubscriptionAdapterRegistrations*. Queste registrazioni descrivono le attuali registrazioni dei subscription adapter per l'oggetto.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registeredSubscriptionAdapters()*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IValidate(Interface):
...     def validate(ob):
...         """Determine whether the object is valid
...
...         Return a string describing a validation problem.
...         An empty string is returned to indicate that the
...         object is valid.
...         """

>>> class IDocument(Interface):
...     summary = Attribute("Document summary")
...     body = Attribute("Document text")

>>> class Document(object):
...     implements(IDocument)
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body

>>> from zope.component import adapts

>>> class AdequateLength(object):
...
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if len(self.doc.body) < 1000:
...             return 'too short'
...         else:
```

```
...             return ''

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerSubscriptionAdapter(AdequateLength, info='ng4')

>>> reg_adapter = list(gsm.registeredSubscriptionAdapters())
>>> 'ng4' in [x.info for x in reg_adapter]
True
```

9.46 registeredUtilities

Restituisce un iterabile di *IUtilityRegistrations*. Queste registrazioni descrivono le attuali registrazioni delle utility per l'oggetto.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registeredUtilities()*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         print "Hello", name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, info='ng5')

>>> reg_adapter = list(gsm.registeredUtilities())
>>> 'ng5' in [x.info for x in reg_adapter]
True
```

9.47 registerHandler

Questa funzione registra un handler. Un handler è un subscriber che non calcola un adapter ma svolge qualche funzione quando viene chiamato.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registerHandler(handler, required=None, name=u'', info='')*
- Vedi anche: *unregisterHandler*

Note: In the current implementation of `zope.component` doesn't support *name* attribute.

Esempio:

```
>>> import datetime

>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()

>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IDocumentCreated(Interface):
...     doc = Attribute("The document that was created")

>>> class DocumentCreated(object):
...     implements(IDocumentCreated)
...
...     def __init__(self, doc):
...         self.doc = doc


>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()

>>> from zope.component import adapter

>>> @adapter(IDocumentCreated)
... def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()


>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerHandler(documentCreated)

>>> from zope.component import handle

>>> handle(DocumentCreated(doc))
>>> doc.created.__class__.__name__
'datetime'
```

9.48 registerSubscriptionAdapter

Questa funzione serve a registrare una factory di subscribers.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registerSubscriptionAdapter(factory, required=None, provides=None, name=u'', info='')*
- Vedi anche: *unregisterSubscriptionAdapter*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
```

```
>>> from zope.interface import implements

>>> class IValidate(Interface):
...     def validate(ob):
...         """Determine whether the object is valid
...
...         Return a string describing a validation problem.
...         An empty string is returned to indicate that the
...         object is valid.
...         """

>>> class IDocument(Interface):
...     summary = Attribute("Document summary")
...     body = Attribute("Document text")

>>> class Document(object):
...     implements(IDocument)
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body

>>> from zope.component import adapts

>>> class AdequateLength(object):
...
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if len(self.doc.body) < 1000:
...             return 'too short'
...         else:
...             return ''

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerSubscriptionAdapter(AdequateLength)
```

9.49 registerUtility

Questa funzione serve a registrare una utility.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *registerUtility(component, provided=None, name=u'', info=u'')*
- Vedi anche: *unregisterUtility*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
```

```
...     def greet(name):
...         "say hello"

>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         print "Hello", name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet)
```

9.50 subscribers

Questa funzione serve a recuperare i subscribers. Vengono restituiti i subscribers che forniscono l'interfaccia passata e che dipendono e sono calcolati dalla sequenza di oggetti richiesti.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *subscribers(required, provided, context=None)*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IValidate(Interface):
...     def validate(ob):
...         """Determine whether the object is valid
...
...         Return a string describing a validation problem.
...         An empty string is returned to indicate that the
...         object is valid.
...         """

>>> class IDocument(Interface):
...     summary = Attribute("Document summary")
...     body = Attribute("Document text")

>>> class Document(object):
...     implements(IDocument)
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body

>>> from zope.component import adapts

>>> class SingleLineSummary:
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
```

```
...
...     def validate(self):
...         if '\n' in self.doc.summary:
...             return 'Summary should only have one line'
...         else:
...             return ''

>>> class AdequateLength(object):
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if len(self.doc.body) < 1000:
...             return 'too short'
...         else:
...             return ''

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerSubscriptionAdapter(SingleLineSummary)
>>> gsm.registerSubscriptionAdapter(AdequateLength)

>>> from zope.component import subscribers

>>> doc = Document("A\nDocument", "blah")
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['Summary should only have one line', 'too short']

>>> doc = Document("A\nDocument", "blah" * 1000)
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['Summary should only have one line']

>>> doc = Document("A Document", "blah")
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['too short']
```

9.51 unregisterAdapter

Questa funzione serve a de-registrare una factory di adapter. Viene restituito un booleano che indica se il registro è stato modificato o meno. La funzione restituisce False se il componente dato è `None` e non ci sono componenti registrati, o se il componente dato non è `None` e non è registrato, altrimenti restituisce True.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *unregisterAdapter(factory=None, required=None, provided=None, name=u'')*
- Vedi anche: *registerAdapter*

Esempio:

```
>>> from zope.interface import Attribute
>>> from zope.interface import Interface

>>> class IDesk(Interface):
...     """A frontdesk will register object's details"""
...
...     def register():
...         """Register object's details"""
...

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...
...     implements(IDesk)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         bookings_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> class Guest(object):
...
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_frontdesk = FrontDeskNG(jack)

>>> IDesk.providedBy(jack_frontdesk)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(FrontDeskNG,
...                     (IGuest,), IDesk, 'ng6')

Si può testare così:

>>> queryAdapter(jack, IDesk, 'ng6')
<FrontDeskNG object at ...>

Ora de-registriamo l'adapter:

>>> gsm.unregisterAdapter(FrontDeskNG, name='ng6')
True
```

```
Dopo la de-registrazione si ha che:

>>> print queryAdapter(jack, IDesk, 'ng6')
None
```

9.52 unregisterHandler

Questa funzione serve a de-registrare un handler. Un handler è un subscriber che non calcola un adapter ma svolge qualche funzione quando viene chiamato. Viene restituito un booleano che indica se il registro è stato modificato o meno.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *unregisterHandler(handler=None, required=None, name=u'')*
- Vedi anche: *registerHandler*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IDocument(Interface):
...
...     summary = Attribute("Document summary")
...     body = Attribute("Document text")

>>> class Document(object):
...
...     implements(IDocument)
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body

>>> doc = Document("A\nDocument", "blah")

>>> class IDocumentAccessed(Interface):
...     doc = Attribute("The document that was accessed")

>>> class DocumentAccessed(object):
...     implements(IDocumentAccessed)
...
...     def __init__(self, doc):
...         self.doc = doc
...         self.doc.count = 0

>>> from zope.component import adapter

>>> @adapter(IDocumentAccessed)
... def documentAccessed(event):
...     event.doc.count = event.doc.count + 1

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerHandler(documentAccessed)
```

```
>>> from zope.component import handle

>>> handle(DocumentAccessed(doc))
>>> doc.count
1

Ora de-registriamo l'handler:

>>> gsm.unregisterHandler(documentAccessed)
True

Dopo la de-registrazione si ha:

>>> handle(DocumentAccessed(doc))
>>> doc.count
0
```

9.53 unregisterSubscriptionAdapter

Questa funzione serve a de-registrare una factory di subscriber. Viene restituito un booleano che indica se il registro è stato modificato o meno. La funzione restituisce False se il componente dato è `None` e non ci sono componenti registrati, o se il componente dato non è `None` e non è registrato, altrimenti restituisce True.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *unregisterSubscriptionAdapter(factory=None, required=None, provides=None, name=u'')*
- Vedi anche: *registerSubscriptionAdapter*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements

>>> class IValidate(Interface):
...     def validate(ob):
...         """Determine whether the object is valid
...
...         Return a string describing a validation problem.
...         An empty string is returned to indicate that the
...         object is valid.
...         """

>>> class IDocument(Interface):
...     summary = Attribute("Document summary")
...     body = Attribute("Document text")

>>> class Document(object):
...     implements(IDocument)
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body

>>> from zope.component import adapts

>>> class AdequateLength(object):
...
```

```
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if len(self.doc.body) < 1000:
...             return 'too short'
...         else:
...             return ''

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> gsm.registerSubscriptionAdapter(AdequateLength)

>>> from zope.component import subscribers

>>> doc = Document("A\nDocument", "blah")
>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
['too short']

Ora de-registriamo il componente:

>>> gsm.unregisterSubscriptionAdapter(AdequateLength)
True

Dopo la de-registrazione si ha:

>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
[]
```

9.54 unregisterUtility

Questa funzione serve a de-registrare una utility. Viene restituito un booleano che indica se il registro è stato modificato o meno. La funzione restituisce False se il componente dato è `None` e non ci sono componenti registrati, o se il componente dato non è `None` e non è registrato, altrimenti restituisce True.

- Posizione: `zope.component - IComponentRegistry`
- Firma: *unregisterUtility(component=None, provided=None, name=u'')*
- Vedi anche: *registerUtility*

Esempio:

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...     def greet(name):
...         "say hello"
```

```
>>> class Greeter(object):
...
...     implements(IGreeter)
...
...     def greet(self, name):
...         return "Hello " + name

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet)

>>> queryUtility(IGreeter).greet('Jack')
'Hello Jack'

Now unregister:

>>> gsm.unregisterUtility(greet)
True

After unregistration:

>>> print queryUtility(IGreeter)
None
```