Le guide complet de l'Architecture de Composants de Zope

Author :	Baiju M
Version :	0.5.8
Printed Book :	http ://www.lulu.com/content/1561045
Online PDF :	http ://www.muthukadan.net/docs/zca.pdf
Traducteur :	Christophe Combelles <ccomb@free.fr>, Stéphane Klein <stephane@harobed.org>

Remerciements

De nombreuses personnes m’ont aidé à écrire ce livre. Le brouillon initial a été relu par mon collègue Brad Allen. Quand j’ai annoncé ce livre sur mon blog, j’ai reçu de nombreux encouragements pour poursuivre ce travail. Kent Tenney a modifié de nombreuses parties du livre et a également réécrit l’application servant d’exemple. De nombreuses autres personnes m’ont envoyé des corrections et des commentaires, dont Lorenzo Gil Sanchez, Michael Haubenwallner, Nando Quintana, Stephane Klein, Tim Cook, Kamal Gill et Thomas Herve. Lorenzo a traduit ce travail en espagnol et Stephane initié la traduction en français. Merci à toutes ces personnes !

Sommaire

1 Mise en route **7**

1.1 Introduction . . . 7
1.2 Petit historique . . . 8
1.3 Installation . . . 8
1.4 Méthode pour expérimenter . . . 9

2 Un exemple **11**

2.1 Introduction . . . 11
2.2 Approche procédurale . . . 11
2.3 Approche orientée objet . . . 12
2.4 Le motif *adaptateur* . . . 13

3 Interfaces **15**

3.1 Introduction . . . 15
3.2 Déclarer des interfaces . . . 16
3.3 Implémentation des interfaces . . . 17
3.4 L'exemple revisité . . . 18
3.5 Interfaces marqueurs . . . 18
3.6 Invariants . . . 19

4 Adaptateurs **21**

4.1 Implémentation . . . 21
4.2 Inscription . . . 22
4.3 récupération d'un adaptateur . . . 23
4.4 Récupérer un adaptateur en utilisant l'interface . . . 24
4.5 Motif adaptateur . . . 24

5 Utilitaires **25**

5.1 Introduction . . . 25
5.2 Utilitaire simple . . . 25
5.3 Utilitaire nommé . . . 26
5.4 Fabrique . . . 27

6 Adaptateurs avancés **29**
6.1 Multi-adaptateur . . . 29
6.2 Adaptateur d'abonnement . . . 30
6.3 Gestionnaire . . . 32

7 Usage de la ZCA dans Zope **35**
7.1 ZCML . . . 35
7.2 Surchargements . . . 36
7.3 NameChooser . . . 37
7.4 LocationPhysicallyLocatable . . . 38
7.5 DefaultSized . . . 38
7.6 ZopeVersionUtility . . . 39

8 Étude de cas **41**
8.1 Introduction . . . 41
8.2 Cas d'utilisation . . . 41
8.3 Survol du code PyGTK . . . 43
8.4 Le code . . . 44
8.5 PySQLite . . . 53
8.6 ZODB . . . 53
8.7 Conclusions . . . 53

9 Reference **55**
9.1 adaptedBy . . . 55
9.2 adapter . . . 55
9.3 adapts . . . 56
9.4 alsoProvides . . . 57
9.5 Attribute . . . 57
9.6 classImplements . . . 58
9.7 classImplementsOnly . . . 58
9.8 classProvides . . . 59
9.9 ComponentLookupError . . . 60
9.10 createObject . . . 60
9.11 Declaration . . . 61
9.12 directlyProvidedBy . . . 61
9.13 directlyProvides . . . 62
9.14 getAdapter . . . 63
9.15 getAdapterInContext . . . 64
9.16 getAdapters . . . 66

9.17 getAllUtilitiesRegisteredFor . . . 66
9.18 getFactoriesFor . . . 67
9.19 getFactoryInterfaces . . . 68
9.20 getGlobalSiteManager . . . 69
9.21 getMultiAdapter . . . 69
9.22 getSiteManager . . . 70
9.23 getUtilitiesFor . . . 71
9.24 getUtility . . . 71
9.25 handle . . . 72
9.26 implementedBy . . . 73
9.27 implementer . . . 74
9.28 implements . . . 74
9.29 implementsOnly . . . 75
9.30 Interface . . . 76
9.31 moduleProvides . . . 76
9.32 noLongerProvides . . . 77
9.33 provideAdapter . . . 78
9.34 provideHandler . . . 78
9.35 provideSubscriptionAdapter . . . 78
9.36 provideUtility . . . 78
9.37 providedBy . . . 78
9.38 queryAdapter . . . 79
9.39 queryAdapterInContext . . . 80
9.40 queryMultiAdapter . . . 82
9.41 queryUtility . . . 83
9.42 registerAdapter . . . 84
9.43 registeredAdapters . . . 85
9.44 registeredHandlers . . . 86
9.45 registeredSubscriptionAdapters . . . 87
9.46 registeredUtilities . . . 88
9.47 registerHandler . . . 89
9.48 registerSubscriptionAdapter . . . 90
9.49 registerUtility . . . 91
9.50 subscribers . . . 91
9.51 unregisterAdapter . . . 93
9.52 unregisterHandler . . . 94
9.53 unregisterSubscriptionAdapter . . . 95
9.54 unregisterUtility . . . 97

Chapitre 1

Mise en route

1.1 Introduction

Le développement de grands systèmes logiciels est toujours une tâche ardue L'expérience montre que l'approche orientée objet est bien appropriée à l'analyse, à la modélisation et à la programmation des systèmes complexes. La conception orientée composants et la programmation basée sur des composants deviennent actuellement très populaires. L'approche basée composants est d'une grande aide pour l'écriture et la maintenance de systèmes logiciels et de leurs tests unitaires. Il existe de nombreux frameworks permettant la conception basée sur des composants dans différents langages, certains étant même indépendants du langage. On peut citer comme exemple : COM de Microsoft, ou XPCOM de Mozilla.

La **Zope Component Architecture (ZCA)** est un framework en Python qui autorise la conception et la programmation basée sur des composants. Elle est très bien adaptée au développement de grands systèmes logiciels. La ZCA n'est pas liée au serveur d'application Zope : elle peut être utilisée pour développer n'importe quelle application en Python. Peut-être devrait-elle s'appeler la « Python Component Architecture ».

Le but de la ZCA est d'utiliser des objets Python de manière efficace. Les composants sont des objets réutilisables fournissant une interface que l'on peut introspecter. Une interface est un objet qui décrit comment on peut travailler avec un composant particulier. Autrement dit, un composant fournit une interface implémentée dans une classe ou tout autre objet appelable (*callable*). La façon dont le composant est implémenté n'a pas d'importance : ce qui compte est que celui-ci soit en accord avec l'interface. Grâce à la ZCA, vous pouvez éclater la complexité d'un système dans plusieurs composants travaillant ensemble. Elle vous aide à créer notamment deux types de composants : les « adaptateurs » et les « utilitaires ».

Trois paquets composent la ZCA :

- `zope.interface` est utilisé pour la définition d'une interface.
- `zope.event` fournit un système simple d'événements.
- `zope.component` sert à la création, l'inscription et la récupération des composants.

Souvenez-vous que la ZCA ne fournit pas de composants par elle-même. Elle n'a pour but que de créer, inscrire et récupérer des composants. Gardez également à l'esprit qu'un adaptateur est une classe Python tout à fait normale (en général une fabrique) et qu'un utilitaire est un simple objet Python appelable.

Le framework ZCA est développé comme une partie du projet Zope 3. Comme mentionné plus haut, c'est un framework purement Python et il peut être utilisé dans n'importe quelle application Python. Actuellement, Zope 3, Zope 2 et Grok l'utilisent abondamment. De nombreux autres projets l'utilisent, y compris des applications non liées au web[1].

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

1.2 Petit historique

Le développement de la ZCA a débuté en 2001 dans le cadre du projet Zope 3. Ce framework est le fruit de l'expérience acquise pendant le développement d'applications d'envergure avec Zope 2. Jim Fulton est le père de ce projet. De nombreuses personnes ont participé à sa conception et à son implémentation, notamment Stephan Richter, Philipp von Weitershausen, Guido van Rossum (*aka. Python BDFL*), Tres Seaver, Phillip J Eby et Martijn Faassen.

À ses débuts, la ZCA définissait des types de composants supplémentaires : les *services* et les *vues*, mais les développeurs ont fini par réaliser qu'un *utilitaire* peut remplacer un *service* et qu'un *multi-adaptateur* peut remplacer une *vue*. Aujourd'hui, la ZCA comporte un nombre très réduit de types de composants de base : les *utilitaires* (en anglais *utilities*), les *adaptateurs* (*adapters*), les *abonnés* (*subscribers*) et les *gestionnaires* (*handlers*). Et encore, les *abonnés* et les *gestionnaires* sont des cas particuliers de l'*adaptateur*.

Pendant le cycle de développement de Zope 3.2, Jim Fulton a proposé une importante simplification de la ZCA[2]. Grâce à cette simplification, une nouvelle interface unique (*IComponentRegistry*) a été créée pour l'inscription des composants globaux et locaux.

Le paquet `zope.component` avait une longue liste de dépendances, dont une grande partie n'était pas nécessaire pour une application non liée à Zope 3. Pendant PyCon 2007, Jim Fulton a ajouté la fonctionnalité *extras_require* à setuptools pour permettre d'extraire le cœur de la ZCA des fonctionnalités annexes[3].

En mars 2009, Tres Seaver a supprimé les dépendances à `zope.deferredimport` et `zope.proxy`.

Aujourd'hui, le projet ZCA est indépendant et possède son propre cycle de publications et son propre dépôt Subversion. Il est fourni avec le framework Zope.[4]. Cependant, les anomalies et les bogues sont toujours pris en charge au travers du projet Zope 3[5] et la liste de diffusion zope-dev est utilisée pour débattre du développement[6]. Il existe également une liste pour les utilisateurs de Zope 3 (*zope3-users*) qui peut être utilisée pour toute demande au sujet de la ZCA[7].

1.3 Installation

Les paquets `zope.component`, `zope.interface` et `zope.event` constituent le cœur de la *Zope Component Architecture*. Ils fournissent des outils permettant de définir, inscrire et rechercher des composants. Le paquet `zope.component` et ses dépendances sont disponibles sous forme d'*egg* dans l'index des paquets Python (PyPI)[8].

Vous pouvez installer `zope.component` et ses dépendances avec *easy_install*[9]

```
$ easy_install zope.component
```

Cette commande télécharge `zope.component` et ses dépendances depuis PyPI et l'installe dans votre *Python path*.

Une autre manière de procéder est de télécharger `zope.component` et ses dépendances depuis PyPI et de les installer. Il faut installer les paquets dans l'ordre donné ci-dessous. Sous Windows, vous aurez probablement besoin du paquet binaire de `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] Dépôt des paquets Python : http ://pypi.python.org/pypi
[9] http ://peak.telecommunity.com/DevCenter/EasyInstall

3. `zope.component`

Pour installer ces paquet, après téléchargement, vous pouvez utiliser la commande `easy_install` avec les eggs comme argument (éventuellement en fournissant tous les eggs sur la même ligne de commande)

```
$ 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
```

Vous pouvez aussi installer ces paquets en les décompressant séparément. Par exemple

```
$ 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
```

Ces méthodes installeront la ZCA dans votre *Python système*, dans le dossier `site-packages`, ce qui peut être problématique. Dans un message sur la liste de diffusion Zope 3, Jim Fulton déconseille d'utiliser le Python système[10]. Vous pouvez utiliser `virtualenv` ou `zc.buildout` pour jouer avec n'importe quel paquet Python, et également pour les déploiements.

1.4 Méthode pour expérimenter

Il existe deux approches populaires en Python pour construire un environnement de travail isolé. La première est `virtualenv` créé par Ian Biking, la deuxième est `zc.buildout` créé par Jim Fulton. Il est même possible d'utiliser ces paquets ensemble. Ils vous aideront à installer `zope.component` et d'autres dépendances dans un environnement de développement isolé. Passer un peu de temps à se familiariser avec ces outils vous sera bénéfique lors des développements et des déploiements.

virtualenv

Vous pouvez installer `virtualenv` grâce à `easy_install`

```
$ easy_install virtualenv
```

Puis vous pouvez créer un nouvel environnement de cette façon

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

Ceci crée un nouvel environnement virtuel dans le dossier `myve`. Maintenant, depuis ce dossier, vous pouvez installer `zope.component` et ses dépendances avec le script `easy_install` situé dans `myve/bin`

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

Ensuite vous pouvez importer `zope.interface` et `zope.component` depuis le nouvel interprète `python` situé dans `myve/bin`

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

Cette commande démarre un interprète Python que vous pourrez utiliser pour lancer le code de ce livre.

zc.buildout

En combinant `zc.buildout` et la recette `zc.recipe.egg`, vous pouvez créer un interprète Python ayant accès aux eggs Python spécifiés. Tout d'abord, installez `zc.buildout` grâce à la commande `easy_install` (vous pouvez le faire à l'intérieur de l'environnement virtuel). Pour créer un nouveau *buildout* servant à expérimenter des eggs Python, commencez par créer un dossier, puis initialisez-le grâce à la commande `buildout init`

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

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

De cette façon, le dossier `mybuildout` devient un « buildout ». Le fichier de configuration de buildout est par défaut *buildout.cfg*. Après initialisation, il doit contenir les lignes suivantes

```
[buildout]
parts =
```

Vous pouvez le modifier pour qu'il corresponde à ceci

```
[buildout]
parts = py

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

Si maintenant vous lancez la commande `buildout` disponible à l'intérieur du dossier `mybuildout/bin`, vous obtiendrez un nouvel interprète Python dans le dossier `mybuildout/bin`

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

Cette commande démarre un interprète Python que vous pourrez utiliser pour lancer le code de ce livre.

Chapitre 2

Un exemple

2.1 Introduction

Imaginez une application servant à enregistrer les clients d'un hôtel. En Python, il est possible de faire ceci de différentes manières. Nous allons commencer par étudier rapidement la méthode procédurale, puis nous diriger vers une approche basique orientée objet. En examinant l'approche orientée objet, nous verrons comment nous pouvons bénéficier des motifs de conception *adaptateur* et *interface*. Cela nous fera entrer dans le monde de la Zope Component Architecture.

2.2 Approche procédurale

Dans toute application professionnelle, le stockage des données est un point critique. Pour une question de simplicité, cet exemple utilise un simple dictionnaire Python comme méthode de stockage. Nous allons générer des identifiants uniques pour le dictionnaire, la valeur associée sera elle-même un dictionnaire contenant les détails de l'enregistrement.

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

Une implémentation minimaliste nécessite une fonction à laquelle on transmet les détails de l'enregistrement et une fonction annexe qui fournit les identifiants uniques utilisés comme clés du dictionnaire de stockage.

Nous pouvons obtenir un identifiant unique de cette façon

```
>>> 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
```

Comme vous pouvez voir, l'implémentation de la fonction *get_next_id* est très simple. La fonction récupère une liste de clés et vérifie si la liste est vide. Si c'est le cas, nous savons que nous en sommes au premier enregistrement et nous renvoyons *1*. Sinon, nous ajoutons *1* à la valeur maximale de la liste et nous renvoyons la nouvelle valeur en résultant.

Ensuite, pour créer des enregistrements dans le dictionnaire bookings_db, nous utilisons la fonction suivante

```
>>> def book_room(name, place) :
...     next_id = get_next_id()
```

```
...     bookings_db[next_id] = {
...     'name' : name,
...     'room' : place
...     }
```

Une application de gestion des enregistrements d'hôtel a besoin de données supplémentaires :

- les numéros de téléphone
- les options des chambres
- les méthodes de paiement
- ...

Et de code pour gérer les données :

- annuler une réservation
- Modifier une réservation
- payer une chambre
- stocker les données
- assurer la sécurité des données
- ...

Si nous devions continuer l'exemple procédural, il faudrait créer plusieurs fonctions, en renvoyant les données de l'une à l'autre. Au fur à mesure que les fonctionnalités seraient ajoutées, le code deviendrait de plus en plus difficile à maintenir et les bogues deviendraient difficiles à trouver et à corriger.

Nous n'irons pas plus loin dans l'approche procédurale. Il sera beaucoup plus facile d'assurer la persistance des données, la flexibilité de la conception et l'écriture de tests unitaires en utilisant des objets.

2.3 Approche orientée objet

Notre discussion sur la conception orientée objet nous amène à la notion de « classe » qui sert à encapsuler les données et le code qui les gère.

Notre classe principale sera « FrontDesk ». Le FrontDesk et d'autres classes auxquelles il délèguera du traitement, sauront comment gérer les données de l'hôtel. Nous allons créer des instances de FrontDesk pour appliquer cette connaissance au métier de la gestion d'hôtel.

L'expérience a montré qu'en consolidant le code et les contraintes sur les données via des objets, on aboutit à un code qui est plus facile à comprendre, à tester et à modifier.

Observons les détails d'implémentation de la classe FrontDesk

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

Dans cette implémentation, l'objet *frontdesk* (une instance de la classe FrontDesk) est capable de prendre en charge les réservations. Nous pouvons l'utiliser de la façon suivante

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

Dans tout projet, on est confronté à des changements de besoins. Dans notre cas, la direction a décidé que chaque client devait fournit un numéro de téléphone, donc nous devons changer le code.

Nous pouvons satisfaire ce nouveau besoin en ajoutant un argument à la méthode *book_room* qui sera ajouté au dictionnaire des valeurs

```
>>> 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
...         }
```

En plus de devoir migrer les données vers le nouveau schéma, nous devons changer tous les appels à la classe `FrontDesk`. Si nous incorporons les coordonnées de chaque client dans un objet et que nous utilisons cet objet pour les réservations, les modifications de code pourront être minimisées. Nous pouvons maintenant changer les détails de l'objet client sans avoir à changer les appels à FrontDesk.

Nous avons donc

```
>>> 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
...         }
```

Nous devons toujours modifier le code pour répondre aux nouvelles demandes. C'est inévitable, cependant notre but est de minimiser ces changements et donc d'améliorer la maintenabilité.

> **Note**
>
> Lors du développement il ne faut jamais hésiter à faire des changements sans craindre de casser l'application. L'avertissement nécessaire doit être immédiatement obtenu grâce à des tests automatisés. Avec des tests bien écrits (et un bon contrôle de versions), vous pouvez faire impunément des changements aussi importants que vous souhaitez. Une bonne source d'informations à propos de cette philosophie de programmation est l'ouvrage *Extreme Programming Explained* par Kent Beck.

En introduisant l'objet guest, vous avez économisé un peu de temps. Mais plus important, l'abstraction apportée par l'objet guest a rendu le système plus simple et mieux compréhensible. Par conséquent, le code est plus facile à restructurer et à maintenir.

2.4 Le motif *adaptateur*

Dans une vraie application, l'objet frontdesk devra gérer des fonctionnalités comme les annulations et les modifications. Avec la conception actuelle, nous devons transmettre l'objet *guest* au frontdesk à chaque fois que nous appelons les méthodes *cancel_booking* et *update_booking*.

Nous pouvons éviter ceci si nous transmettons l'objet *guest* à FrontDesk.__init__() et si nous le stockons en attribut de l'instance.

```
>>> 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
...         }
...
...     def cancel_booking(self):
...         guest = self.guest
...         #code for cancellations goes here ...
...
...     def update_booking(self):
...         guest = self.guest
...         #code for updatiion goes here ...
```

La solution que nous avons obtenue est un motif de conception courant appelé *adaptateur*. Le *Gang of Four*[12] résume ainsi l'*esprit* de l'adaptateur

> « Convertir l'interface d'une classe en une autre interface à laquelle le client s'attend. L'adaptateur permet à des classes de fonctionner ensemble même si elles ont des interfaces incompatibles. »

En général, un objet adaptateur *contient* un objet adapté

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

Ce motif sera utile lorsqu'on sera confronté à des détails d'implémentation qui dépendent de considérations telles que :
– modification des besoins client
– modification des méthodes de stockage (ZODB, RDBM, XML)
– modification des méthodes de sortie (HTML, PDF, texte pur)
– modification de la source utilisée (ReST, Markdown, Textile)
La ZCA utilise des adaptateurs et un *registre de composants* pour fournir la capacité de changer les détails d'implémentation du code via de la *configuration*.

Comme nous pouvons le constater dans la section sur les adaptateurs de la ZCA, la capacité de configurer les détails d'implémentation fournit des avantages intéressants :
– on peut interchanger les implémentations
– on peut ajouter de nouvelles implémentations si besoin
– on améliore les possibilités de réutilisation, aussi bien d'un code existant que du code utilisant la ZCA.
Ces possibilités mènent à du code qui est flexible, évolutif et réutilisable. Il y a un cependant un coût : maintenir le registre de composants ajoute un niveau de complexité à l'application. Si une application n'a pas besoin de ces avantages, la ZCA n'est pas pas nécessaire.

Nous sommes maintenant prêts à étudier la Zope Component Architecture, en commençant avec les interfaces.

Chapitre 3

Interfaces

3.1 Introduction

Le fichier README.txt[11] dans chemin/vers/zope/interface définit les interfaces de cette manière

```
Les interfaces sont des objets qui spécifient (documentent) le
comportement externe des objets qui les « fournissent ».
Une interface spécifie un comportement au travers :

- de documentation informelle dans une docstring

- de définitions d'attributs

- d'invariants qui sont des conditions posées sur les objets
fournissant l'interface.
```

L'ouvrage de référence *Design Patterns*[12] par le *Gang of Four* recommande que vous devez « programmer pour une interface, pas pour une implémentation ». Définir une interface formelle est utile dans la compréhension d'un système. De plus, les interfaces vous apportent tous les bénéfices de la ZCA.

Une interface spécifie les caractéristiques d'un objet, son comportement et ses capacités. Elle décrit le « quoi » d'un objet. Pour apprendre le « comment », vous devez regarder l'implémentation.

Les métaphores couramment utilisées pour les interfaces sont *contrat* ou *plan*, des termes légaux et architecturaux pour représenter un jeu de spécifications.

Dans certains langages de programmation comme Java, C# ou VB.NET, les interfaces sont un aspect explicite du langage. Étant donné que Python ne possède pas nativement d'interfaces, la ZCA les implémente comme des meta-classes desquelles on hérite.

Voici un exemple classique de Hello World

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

[11] L'arborescence du code de Zope contient de nombreux fichiers README.txt qui offrent une très bonne documentation.
[12] http ://en.wikipedia.org/wiki/Design_Patterns

Dans la classe ci-dessus, on a défini une méthode *goodmorning*. Si vous appelez cette méthode depuis un objet créé en utilisant cette classe, vous obtiendrez un *Good morning, ... !*

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

Ici, `host` est l'objet réel que votre code utilise. Si vous voulez examiner les détails d'implémentation, vous devez accéder à la classe `Host`, soit via le code source, soit au travers d'un outil de documentation d'API[13].

Maintenant nous allons commencer à utiliser les interfaces de la ZCA. Pour la classe ci-dessus, vous pouvez définir l'interface comme suit

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

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

Vous pouvez constater que l'interface hérite de zope.interface.Interface. C'est de cette façon (abusive ?) que la ZCA définit des interfaces. Le préfixe « I » pour le nom de la classe est une convention utile.

3.2 Déclarer des interfaces

Vous avez déjà vu comment déclarer une interface en utilisant `zope.interface` dans la section précédente. Cette section va expliquer les concepts en détail.

Prenez cette exemple d'interface

```
>>> 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'interface `IHost` possède deux attributs, `name` et `goodmorning`. Rappelez-vous qu'en Python les méthodes des classes sont aussi des attributs. L'attribut `name` est défini en utilisant la classe `zope.interface.Attribute`. Quand vous ajoutez l'attribut `name` à l'interface `IHost`, vous ne définissez pas de valeur initiale. La raison de définir l'attribut `name` ici est simplement d'indiquer que toute implémentation de cette interface doit fournir un attribut nommé `name`. Dans ce cas, vous n'indiquez même pas de quel type doit être l'attribut ! Vous pouvez juste fournir une chaîne de documentation comme premier argument à `Attribute`.

L'autre attribut, `goodmorning` est une méthode définie en utilisant une fonction. Notez bien que `self` n'est pas nécessaire dans les interfaces, car `self` est un détail d'implémentation de la classe. Il est possible pour un module d'implémenter cette interface. Si un module implémente cette interface, cela signifiera qu'un attribut `name` et une fonction `goodmorning` seront définis. Et la fonction `goodmorning` devra accepter un argument.

Nous allons maintenant voir comment effectuer le lien entre les objets, les classes et les interfaces. Seuls les objets sont vivants : ce sont des instances de classes. Et les interfaces représentent la définition des objets, donc les classes ne sont qu'un détail d'implémentation. C'est pour cette raison que vous devez programmer pour une interface, pas pour une implémentation.

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

Nous devons nous familiariser avec deux termes supplémentaires pour comprendre les autres concepts. Le premier est *fournir*, le deuxième est *implémenter*. Les objets fournissent des interfaces, tandis que les classes implémentent des interfaces. Autrement dit, les objets fournissent les interfaces que les classes implémentent. Dans l'exemple ci-dessus, l'objet `host` fournit l'interface `IHost` et la classe `Host` implémente l'interface `IHost`. Un objet peut fournir plusieurs interfaces. Les objets peuvent également fournir directement des interfaces, en plus de celles implémentées par leur classe.

> **Note**
>
> Les classes sont les détails d'implémentation des objets. En Python, les classes sont des objets appelables (*callable*). Pourquoi un autre objet appelable ne pourrait-il pas implémenter une interface ? En fait c'est possible. Pour n'importe quel objet appelable, vous pouvez déclarer qu'il produit des objets qui fournissent une interface donnée, en disant simplement que cet objet appelable implémente l'interface. Ces objets appelables sont généralement nommés des « fabriques ». Étant donné que les fonctions sont des objets appelables, une fonction peut implémenter une interface.

3.3 Implémentation des interfaces

Pour déclarer qu'une classe implémente une interface donnée, utilisez la fonction `zope.interface.implements` dans la déclaration de classe.

Considérons cet exemple. Ici, `Host` implémente `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**
>
> Si vous vous demandez comment `implements` fonctionne, faites un tour sur l'article de blog de James Henstridge (http ://blogs.gnome.org/jamesh/2005/09/08/python-class-advisors/) . Dans la section sur les adaptateurs, vous allez voir une fonction `adapts`, celle-ci fonctionne de manière similaire.

Comme `Host` implémente `IHost`, les instances de `Host` fournissent `IHost`. Il existe des méthodes permettant d'introspecter les déclarations. La déclaration peut également s'écrire en dehors de la classe. Au lieu d'écrire `interface.implements(IHost)` comme dans l'exemple ci-dessus, vous pouvez écrire la chose suivante après la déclaration de classe

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

3.4 L'exemple revisité

Maintenant retournons à notre application exemple. Voici comment définir l'interface de l'objet frontdesk

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

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

Nous avons d'abord importé la classe `Interface` depuis le module `zope.interface`. Si vous définissez une sous-classe de cette classe `Interface`, ce sera considéré comme une interface, du point de vue de l'Architecture de Composants. Une interface peut être implémentée, comme nous l'avons vu, dans une classe ou tout autre objet appelable.

L'interface de frontdesk ici définie est `IDesk`. La chaîne de documentation de l'interface donne une idée sur la nature de l'objet. En définissant une méthode dans l'interface, vous avez passé un contrat avec le composant, imposant la présence d'une méthode du même nom. Dans la déclaration de la méthode côté interface, le premier argument ne doit pas être *self*, car une interface ne sera jamais instanciée et ses méthodes ne seront jamais appelées. Le rôle d'une interface est simplement de documenter quels arguments et quelles méthodes doivent apparaître dans une classe qui l'implémente, et le paramètre *self* n'est qu'un détail d'implémentation qui n'a pas besoin d'être documenté.

Comme vous le savez, une interface peut aussi spécifier des attributs normaux

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

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

Dans cette interface, l'objet guest a deux attributs contenant de la documentation. Une interface peut spécifier à la fois des méthodes et des attributs. Une interface peut être implémentée dans une classe, un module ou tout autre objet. Par exemple une fonction peut créer dynamiquement le composant et le renvoyer, auquel cas on dit que la fonction implémente l'interface.

Vous savez maintenant ce qu'est une interface, comment la créer et l'utiliser. Dans le chapitre suivant nous allons voir comment une interface peut être utilisée pour définir un composant adaptateur.

3.5 Interfaces marqueurs

Une interface peut être utilisée pour déclarer qu'un objet appartient à un type donné. Une interface sans aucun attribut ni aucune méthode est appelée une « interface marqueur ».

Voici une *interface marqueur*

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

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

Cette interface peut être utilisée pour déclarer qu'un objet est un client spécial.

3.6 Invariants

Parfois vous aurez besoin de définir des règles ou des contraintes sur les attributs de vos composants. Ces types de règles sont appelés des *invariants*. Vous pouvez utiliser `zope.interface.invariant` pour définir des `invariants` sur vos objets dans leur interface.

Considérons un exemple simple, avec un objet *person*. Une objet *person* a un attribut *nom*, un attribut *email* et un attribut *phone*. Comment peut-on implémenter une règle de validation qui oblige à définir au choix le *phone* ou l'*email* mais pas forcément les deux ?

Créez tout d'abord un objet appelable, soit une simple fonction soit une instance appelable d'une classe de la façon suivante

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

Ensuite définissez l'interface de l'objet *person* en utilisant la fonction `zope.interface.invariant` pour définir l'invariant

```
>>> 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)
```

Maintenant, utilisez la méthode *validateInvariants* de l'interface pour effectuer la validation

```
>>> 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
```

Vous constatez que l'ojet *jack* est validé sans erreur. Mais l'objet *jill* n'a pas pu valider la contrainte de l'invariant, il a donc levé une exception.

Chapitre 4

Adaptateurs

4.1 Implémentation

Cette section décrit les adaptateurs en détail. L'Architecture de Composants, comme vous l'avez remarqué, fournit une aide dans l'utilisation efficace des objets Python. Les composants adaptateurs sont l'un des composants basiques utilisés par l'Architecture de Composants de Zope pour utiliser efficacement des objets Python. Les adaptateurs sont aussi des objets Python, mais avec une interface bien définie.

Pour déclarer qu'une classe est un adaptateur, utilisez la fonction *adapts* définie dans le paquet `zope.component`. Voici un nouvel adaptateur *FrontDeskNG* avec une déclaration explicite d'interface

```
>>> 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
...         }
```

Ce que nous avons défini ici est un *adaptateur* pour *IDesk*, qui s'adapte à l'objet *IGuest*. L'interface *IDesk* est implémentée par la classe *FrontDeskNG*. Donc une instance de cette classe fournira l'interface *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
```

FrontDeskNG est juste un adaptateur que nous avons créé. Vous pouvez créer d'autres adaptateurs qui prendront en charge le bureau d'enregistrement différemment.

4.2 Inscription

Pour utiliser ce composant adaptateur, vous devez l'inscrire dans un registre de composants (appelé également « gestionnaire de site »). Un gestionnaire de site réside normalement à l'intérieur d'un site. Le site et le gestionnaire de site prendront leur importance lors du développement d'une application Zope 3. Pour l'instant vous avez juste besoin de connaître les notions de site global et de gestionnaire global de site (ou registre de composant). Un gestionnaire global de site est situé en mémoire, alors qu'un gestionnaire de site local est persistant.

Pour inscrire votre composant, commencez par récupérer le gestionnaire global de site

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

Pour récupérer le gestionnaire global de site, vous devez appeler la fonction `getGlobalSiteManager`` disponible dans le paquet ``zope.component`. En fait, le gestionnaire global de site est disponible dans un attribut (`globalSiteManager`) du paquet `zope.component`. Vous pouvez donc utiliser directement l'attribut `zope.component.globalSiteManager`. Pour inscrire l'adaptateur comme un composant, utilisez la méthode `registerAdapter` du registre de composants. Le premier argument doit être votre classe ou fabrique d'adaptateur. Le deuxième argument est un tuple d'objets adaptés, c'est à dire les objets sur lesquels vous vous adaptez. Dans cet exemple, vous vous adaptez seulement à l'objet *IGuest*. Le troisième argument est l'interface implémentée par le composant adaptateur. Le quatrième argument est optionnel, il s'agit du nom de cet adaptateur particulier. Si vous donnez un nom à l'adaptateur, celui devient un *adaptateur nommé*. Si aucun nom n'est donné, la valeur transmise par défaut est une chaîne vide (").

Dans l'inscription ci-dessus, vous avez donné l'interface de l'objet adapté et l'interface fournie par l'adaptateur. Comme vous avez déjà donné ces informations dans l'implémentation de l'adaptateur, il est inutile de les spécifier à nouveau. En réalité, vous pouvez effectuer l'inscription de la manière suivante

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

Pour effectuer l'inscription, il existe d'anciennes API que vous devriez éviter. Les fonctions de l'ancienne API commencent par *provide*, par exemple : `provideAdapter`, `provideUtility`, etc. Si vous développez une application Zope 3, vous pouvez utiliser le langage ZCML (Zope Configuration Markup Language) pour effectuer les inscriptions des composants. Avec Zope 3, les composants locaux (persistants) peut être inscrits depuis la ZMI (Zope Management Interface), ou bien par programmation.

Vous avez inscrit *FrontDeskNG* avec le nom *ng*. De la même manière, vous pouvez inscrire d'autre adaptateurs avec différents noms. Si un composant est inscrit sans nom, son nom sera la chaîne vide par défaut.

> **Note**
>
> Les composants locaux sont persistants mais les composants globaux sont en mémoire. Les composants globaux sont inscrits en fonction de la configuration de l'application. Les composants locaux sont récupérés dans la base de données au démarrage de l'application.

4.3 récupération d'un adaptateur

La récupération des composants inscrits dans le registre est effectuée grâce à deux fonctions disponibles dans le paquet `zope.component`. La première est `getAdapter`, la deuxième `queryAdapter`. Les deux fonctions prennent les mêmes arguments. `getAdapter` lève une exception `ComponentLookupError` si la recherche de composant échoue, tandis que `queryAdapter` renvoie *None*.

Vous pouvez importer ces fonctions comme ceci

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

Dans la section précédente, nous avons inscrit un composant qui fournit l'interface *IDesk* avec un nom *ng*, et qui s'adapte à l'objet guest. Dans la première section nous avons créé un objet nommé *jack*.

Voici maintenant comment récupérer un composant qui s'adapte à l'interface de l'objet *jack* (*IGuest*) et qui fournit l'interface *IDesk* également avec le nom *ng*. Ici, `getAdapter` et `queryAdapter` fonctionnent de la même manière

```
>>> getAdapter(jack, IDesk, 'ng') #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
>>> queryAdapter(jack, IDesk, 'ng') #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
```

Vous pouvez constater que le premier argument doit être l'objet adapté, puis l'interface qui doit être fournie par le composant et finalement le nom du composant adaptateur.

Si vous essayez de récupérer le composant grâce à un nom inutilisé pour l'inscription mais pour le même objet adapté et la même interface, la recherche échouera. Voici dans ce cas comment fonctionnent les deux méthodes

```
>>> getAdapter(jack, IDesk, 'not-exists') #doctest : +ELLIPSIS
Traceback (most recent call last) :
...
ComponentLookupError : ...
>>> reg = queryAdapter(jack,
...           IDesk, 'not-exists') #doctest : +ELLIPSIS
>>> reg is None
True
```

`getAdapter` a levé une exception `ComponentLookupError` mais `queryAdapter` a renvoyé *None*.

Le troisième argument, le nom d'inscription, est optionnel. Si le troisième argument n'est pas fourni, sa valeur par défaut est la chaîne vide (''). Comme il n'y a aucun composant inscrit avec la chaîne vide comme nom, `getAdapter` lève l'exception `ComponentLookupError`. De la même manière, `queryAdapter` renvoie *None*. Voyez vous-même comment cela fonctionne

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

Dans cette section, vous avez appris à inscrire un simple adaptateur et à le récupérer depuis le registre de composants. Ce type d'adaptateurs est appelé un adaptateur simple, car il ne s'adapte qu'à un seul objet. Si un adaptateur s'adapte à plusieurs objets, on l'appelle un multi-adaptateur.

4.4 Récupérer un adaptateur en utilisant l'interface

Les adaptateurs peuvent être récupérés directement en utilisant les interfaces, mais cela ne fonctionne qu'avec les adaptateurs simples. Le premier argument est l'objet adapté, le deuxième argument est un argument mot-clé. Si la recheche d'adaptateur échoue, le deuxième argument est renvoyé.

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

Le mot-clé peut être omis :

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

Si le deuxième argument n'est pas fourni, une erreur *TypeError* est levée :

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

Ici, *FrontDeskNG* est inscrit sans nom :

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

Maintenant la récupération de l'adaptateur doit réussir :

```
>>> IDesk(jack, 'default-output') #doctest: +ELLIPSIS
<FrontDeskNG object at ...>
```

Pour les cas simples, vous pouvez utiliser l'interface pour récupérer des adaptateurs.

4.5 Motif adaptateur

Le principe de l'adaptateur dans l'Architecture de Composants de Zope est très similaire au *motif adaptateur* classique, tel que décrit dans le livre « Design Patterns ». Mais l'objectif visé par l'utilisation des adaptateurs dans la ZCA est plus large que le principe de l'adaptateur lui-même. Le but de l'adaptateur est de convertir l'interface d'une classe en une autre interface attendue par le client. Ceci permet de faire fonctionner des classes ensemble même si elles ont des interfaces incompatibles entre elles. Mais dans la section *motivations* du livre « Design Patterns », le GoF dit : « souvent, l'adaptateur est responsable d'une fonctionnalité que la classe adaptée ne fournit pas ». Un adaptateur de la ZCA se concentre effectivement plus sur l'ajout de fonctionnalités que sur la création d'une nouvelle interface pour un objet adapté. L'adaptateur de la ZCA permet aux classes adaptateurs d'étendre des fonctionnalités par ajout de méthodes. (Il peut être intéressant de remarquer que l'adaptateur était appelé *Fonctionnalité* (*feature*) dans les premières étapes de conception de la ZCA.)[14]

La citation du Gang of Four dans le paragraphe ci-dessus se termine de la façon suivante : « ... que la classe adaptée ne fournit pas ». Mais dans la phrase suivante nous avons utilisé « objet adapté » plutôt que « classe adaptée », car le GoF décrit deux variantes d'adaptateurs selon les implémentations. Le premier est appelé *adaptateur de classe*, le deuxième *adaptateur d'objet*. Un adaptateur de classe utilise l'héritage multiple pour adapter une interface à une autre, alors que l'adaptateur d'objet se base sur la composition d'objet. L'adaptateur de la ZCA utilise le principe de l'adaptateur d'objet, qui utilise la délégation comme mécanisme de composition. Le second principe du GoF à propos de la conception orientée objets est la suivante : « favorisez la composition d'objets plutôt que l'héritage de classes ». Pour approfondir le sujet, reportez-vous au livre « Design Patterns ».

Les principaux atouts des adaptateurs de la ZCA sont les interfaces explicites et les registres de composants. Les composants adaptateurs de la ZCA sont inscrits dans le registre de composants et sont récupérés par les objets clients via leur interface et leur nom si besoin.

[14] Thread discussing renaming of *Feature* to *Adapter* : http ://mail.zope.org/pipermail/zope3-dev/2001-December/000008.html

Chapitre 5

Utilitaires

5.1 Introduction

Vous connaissez maintenant le principe des interfaces, des adaptateurs et du registre de composants. Parfois, il peut être utile d'inscrire un objet qui ne s'adapte à rien du tout, par exemple une connexion à une base de données, un analyseur XML ou un objet retournant des identifiants uniques. Ce type de composant fourni par la ZCA est appelé un « utilitaire » (*utility*).

Les utilitaires sont simplement des objets qui fournissent une interface et qui sont récupérés à partir de cette interface et d'un nom. Cette approche permet de créer un registre global dans lequel des instances peuvent être inscrites et récupérées à différents endroits de votre application, sans avoir besoin de transmettre les instances comme paramètres.

Vous n'avez pas besoin d'inscrire toutes les instances de composants de cette façon. Inscrivez seulement les composants que vous voulez rendre interchangeables.

5.2 Utilitaire simple

Un utilitaire peut être inscrit avec ou sans nom. Un utilitaire inscrit avec un nom est appelé un *utilitaire nommé*. Vous le verrez dans la prochaine section. Avant d'implémenter l'utilitaire, comme d'habitude, définissez son interface. Voici une interface qui dit bonjour

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

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

Comme un adaptateur, un utilitaire peut avoir plusieurs implémentations. Voici une implémentation possible de l'interface ci-dessus

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

L'utilitaire réel sera une instance de cette classe. Pour utiliser cet utilitaire, vous devez l'inscrire, puis vous pourrez la récupérer en utilisant l'API de la ZCA. Vous pouvez inscrire une instance de cette classe (*utilitaire*) grâce à `registerUtility`

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

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

Dans cet exemple, vous avez inscrit l'utilitaire en fournissant l'interface *IGreeter*. Vous pouvez récupérer l'utilitaire avec *queryUtility* ou *getUtility*

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

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

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

Comme vous pouvez le constater, alors que les adaptateurs sont habituellement des classes, les utilitaires sont habituellement des instances de classes. vous ne créez une instance d'utilitaire qu'une fois, alors que les instances d'adaptateurs sont créées à chaque fois que vous les récupérez.

5.3 Utilitaire nommé

Lorsque vous inscrivez un composant utilitaire, de la même manière que pour les adaptateurs, vous pouvez utiliser un nom. Comme mentionné dans la section précédente, un utilitaire inscrit avec un nom est appelé un *utilitaire nommé*.

Voici comment inscrire l'utilitaire *greeter* avec un nom

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

Dans cet exemple, nous avons inscrit l'utilitaire avec un nom en fournissant l'interface *IGreeter*. Vous pouvez récupérer l'utilitaire avec *queryUtility* ou *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'
```

Remarquez que vous devez utiliser le nom de l'utilitaire comme second argument pour que la récupération fonctionne.

Appeler *getUtility* sans nom (comme second argument) est équivalent à l'appeler avec un nom vide (''), car la valeur par défaut pour ce second argument est justement la chaîne vide. Donc le mécanisme de recherche de composant essaiera de trouver le composant ayant une chaîne vide comme nom et il échouera. Lorsqu'une recherche de composant échoue, le résultat est une exception `ComponentLookupError`. Souvenez-vous qu'aucun composant aléatoire avec un autre nom ne sera renvoyé. Les fonctions de récupération d'adaptateurs, *getAdapter* et *queryUtility* fonctionnent de manière similaire.

5.4 Fabrique

Une `fabrique` est un composant utilitaire qui fournit l'interface `IFactory`.

Pour créer une fabrique, commencez par définir l'interface de l'objet

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

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

Voici une implémentation factice de l'interface *IDatabase*

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

Vous pouvez créer une fabrique en utilisant `zope.component.factory.Factory`

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

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

Maintenant vous pouvez l'inscrire de cette façon

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

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

Pour utiliser la fabrique, vous pouvez faire comme ceci

```
>>> from zope.component import queryUtility
>>> queryUtility(IFactory, 'fakedb')() #doctest : +ELLIPSIS
<FakeDb object at ...>
```

Voici l'écriture simplifiée pour utiliser une fabrique

```
>>> from zope.component import createObject
>>> createObject('fakedb') #doctest : +ELLIPSIS
<FakeDb object at ...>
```

Chapitre 6

Adaptateurs avancés

Ce chapître traite de concepts avancés comme les multi-adaptateurs, les abonnés (subscribers) et les gestionnaires (handles).

6.1 Multi-adaptateur

Un adaptateur simple s'adapte normalement à un seul objet, mais un adaptateur peut s'adapter à plusieurs objets. Si un adaptateur s'adapte à plus d'un objet, il s'appelle un multi-adaptateur.

```
>>> 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) #doctest : +ELLIPSIS
<MyFunctionality object at ...>

>>> myfunctionality = getMultiAdapter((one,two), IFunctionality)
>>> myfunctionality.one #doctest : +ELLIPSIS
<One object at ...>
>>> myfunctionality.two #doctest : +ELLIPSIS
<Two object at ...>
```

6.2 Adaptateur d'abonnement

À la différence des adaptateurs habituels, les adaptateurs d'abonnement sont utilisés quand on veut récupérer tous les adaptateurs qui adaptent un objet à une interface donnée. Un adaptateur d'abonnement est également appelé *abonné*.

Considérons un problème de validation. Nous avons des objets et nous voulons vérifier s'ils satisfont à un certain critère. Nous définissons une interface de validation

```
>>> 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.
...         """
```

Nous avons également des documents

```
>>> 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
```

Maintenant, nous pouvons avoir besoin de préciser différentes règles de validation pour les documents. Par exemple, nous voulons que le résumé tienne sur une seule ligne

```
>>> 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 ''
```

Ou bien nous voulons que le corps du document fasse au moins 1000 caractères

```
>>> 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 ''
```

Nous pouvons inscrire ces deux composants comme des adaptateurs d'abonnement

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

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

Nous pouvons ensuite utiliser ces abonnés pour valider les objets

```
>>> 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 Gestionnaire

Les gestionnaires sont des fabriques d'abonnés qui ne produisent rien du tout. Ils font la totalité de leur travail lorsqu'ils sont appelés. Les gestionnaires sont typiquement utilisés pour gérer des événements. Les gestionnaires sont aussi connus sous le nom d'`abonnés à des événements` ou `adaptateurs d'abonnement à des événements`.

Les abonnés à des événements sont différents des autres abonnés car l'objet qui appelle les abonnés ne s'attend pas à interagir avec eux d'une quelconque façon. Par exemple, un publicateur d'événements ne s'attend pas à récupérer une valeur de retour. Comme les abonnés n'ont pas besoin de fournir une API aux objets qui les appellent, il est plus naturel de les définir avec des fonctions, plutôt qu'avec des classes. Par exemple, dans un système de gestion de documents, nous pouvons avoir besoin de gérer les heures de création des documents

```
>>> import datetime

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

Dans cet exemple, nous avons une fonction qui prend un événement et effectue un traitement. La fonction ne retourne en fait rien du tout. C'est un cas particulier de l'adaptateur d'abonnement qui s'adapte à un événement et ne fournit rien. Tout le travail est effectué lorsque la fabrique de l'adaptateur est appelée. Nous appelons les abonnés qui ne fabriquent rien des « gestionnaires ». Il existe des APIs spécialisées dans leur inscription et leur récupération.

Pour inscrire l'abonné ci-dessus, nous définissons un événement « document créé »

```
>>> 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
```

Nous modifions également notre définition du gestionnaire

```
>>> 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()
```

Ceci marque le gestionnaire comme étant un adaptateur des événements *IDocumentCreated*.

Nous inscrivons le gestionnaire

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

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

Il est maintenant possible de créer un événement et d'utiliser la fonction *handle* pour appeler les gestionnaires inscrits à l'événement

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

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

Chapitre 7

Usage de la ZCA dans Zope

L'Architecture de Composants de Zope est utilisée dans Zope 2 et dans Zope 3. Ce chapitre traite de l'usage de la ZCA dans Zope.

7.1 ZCML

Le **Zope Configuration Markup Language (ZCML)** est un système de configuration pour l'inscription des composants. Au lieu d'utiliser l'API Python pour les inscriptions, vous pouvez utiliser le ZCML. Pour pouvoir utiliser le ZCML, vous devez installer des paquets supplémentaires.

Vous pouvez installer `zope.component` avec la prise en charge du ZCML en utilisant easy_install comme ceci

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

Un fichier ZCML doit commencer par une directive `configure` avec la déclaration d'espace de nom appropriée

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

La directive *adapter* peut être utilisée pour inscrire des adaptateurs

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

Les attributs *provides* et *for* sont optionnels, à condition que vous ayez fait les déclaration correspondantes dans l'implémentation

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

Si vous voulez inscrire le composant comme un adaptateur nommé, vous pouvez utiliser l'attribut *name*

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

Un utilitaire peut être inscrit avec la directive *utility*

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

L'attribut *provides* est optionnel, à condition que vous ayez fait la déclaration correspondante dans l'implémentation

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

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

Si vous voulez inscrire le composant comme un utilitaire nommé, vous pouvez utiliser l'attribut *name*

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

Plutôt que d'utiliser directement le composant, vous pouvez aussi fournir une fabrique

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

7.2 Surchargements

Lorsque vous inscrivez des composants avec l'API Python (méthodes `register*`), le dernier composant inscrit remplace le précédent, si les deux sont inscrits avec les mêmes arguments. Considérons l'exemple suivant

```
>>> 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) #doctest : +ELLIPSIS
<AP object at ...>
```

Si vous inscrivez un autre adaptateur, celui existant sera surchargé

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

>>> getAdapter(a, IP) #doctest : +ELLIPSIS
<AP2 object at ...>
```

Mais lorsque vous inscrivez des composants en ZCML, la deuxième inscription provoquera un conflit. Ceci est fait pour éviter les surchargements accidentels d'inscriptions, qui sont difficiles à retrouver et corriger. L'utilisation de ZCML est donc un avantage pour votre application.

Parfois vous aurez besoin de surcharger une inscription existante. ZCML fournit une directive `includeOverrides` à cet effet. Avec la directive ci-dessous, vous pouvez écrire vos surcharges dans un fichier séparé

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

7.3 NameChooser

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

C'est un adaptateur qui choisit un nom unique pour un objet dans un conteneur.

L'inscription de l'adaptateur ressemble à ceci

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

Dans cette inscription, vous pouvez observer qu'on s'adapte à un `IWriteContainer` et que l'adaptateur fournit `INameChooser`.

Cet adaptateur fournit une fonctionnalité très pratique pour les développeurs Zope. Les implémentations principales de `IWriteContainer` dans Zope 3 sont *zope.app.container.BTreeContainer*‘ et `zope.app.folder.Folder`. Normalement vous hériterez de ces implémentations pour créer vos propres classes de conteneurs. Supposez qu'il n'y ait pas d'interface `INameChooser` ni d'adaptateur : vous seriez obligé de recréer cette fonctionnalité séparément pour chaque implémentation.

7.4 LocationPhysicallyLocatable

Emplacement : `zope.location.traversing.LocationPhysicallyLocatable`

Cet adaptateur est fréquemment utilisé dans les applications Zope 3, mais habituellement il est appelé au travers d'une API dans `zope.traversing.api`. (Certains vieux codes utilisent même les fonctions `zope.app.zapi` qui ne sont qu'une indirection supplémentaire).

L'inscription de l'adaptateur ressemble à ceci

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

L'interface fournie et l'interface adaptée sont données dans l'implémentation, dont voici le début

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

Normalement, presque tous les objets persistants d'une application Zope 3 fournissent l'interface `ILocation`. Cette interface n'a que deux attributs, `__parent__` et `__name__`. `__parent__` est le parent dans la hiérarchie des objets et `__name__` est le nom, vu depuis le parent.

L'interface `IPhysicallyLocatable` a quatre méthodes : `getRoot`, `getPath`, `getName`, et `getNearestSite`.

- `getRoot` renvoie l'objet racine physique.
- `getPath` renvoie une chaîne donnant le chemin physique vers l'objet.
- `getName` renvoie le dernier segment du chemin physique.
- `getNearestSite` renvoie le site dans lequel l'objet est contenu. Si l'objet est lui-même un site, il est renvoyé.

Si vous apprenez Zope 3, ce sont des choses importantes dont vous aurez besoin très souvent. Pour comprendre la beauté de ce système, vous devriez regarder comment Zope 2 récupère l'objet racine physique et comment ceci est implémenté. Il existe une méthode `getPhysicalRoot` virtuellement pour tous les objets conteneurs.

7.5 DefaultSized

Emplacement : `zope.size.DefaultSized`

Cet adaptateur n'est qu'une implémentation par défaut de l'interface `ISized`. Il est inscrit pour tous les types d'objets. Si vous voulez inscrire cet adaptateur pour une interface particulière, vous devez surcharger cette inscription avec votre implémentation.

L'inscription de l'adaptateur ressemble à ceci

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

Comme vous pouvez voir, l'interface adaptée est « * », pour pouvoir s'adapter à tous les types d'objets.

`ISized` est une interface simple possédant deux méthodes

```
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.
        """
```

Vous pouvez trouver un autre adaptateur `ISized` pour `IZPTPage` dans le paquet `zope.app.zptpage`.

7.6 ZopeVersionUtility

Emplacement : `zope.app.applicationcontrol.ZopeVersionUtility`

Cet utilitaire donne la version de Zope en fonctionnement.

L'inscription est la suivante

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

L'interface fournie, `IZopeVersion`, n'a qu'une méthode nommée `getZopeVersion`. Cette méthode renvoie une chaîne contenant la version de Zope (avec éventuellement un changeset SVN).

L'implémentation par défaut, `ZopeVersionUtility`, récupère le numéro de version dans un fichier `version.txt` du répertoire *zope/app*. Si Zope fonctionne depuis un espace de travail Subversion, il montre le dernier numéro de révision. Si aucun des deux ne fonctionne il renvoie *DevCenter/Unknown*.

Chapitre 8

Étude de cas

> **Note**
> Ce chapitre n'est pas terminé. Merci d'envoyer vos suggestions !

8.1 Introduction

Ce chapitre est un exemple de création d'une application lourde avec la bibliothèque d'interface graphique PyGTK et la ZCA. Cette application utilise également deux types différents de mécanismes de persistance des données : le premier est une base de données objet (ZODB), l'autre est une base de données relationnelle (SQLite). Cependant, en pratique, un seul des deux mécanismes peut être utilisé dans une installation donnée. L'utilisation de deux types de persistance est là pour démontrer comment utiliser la ZCA pour brancher des composants entre eux. La majorité de code de cette application est lié à PyGTK.

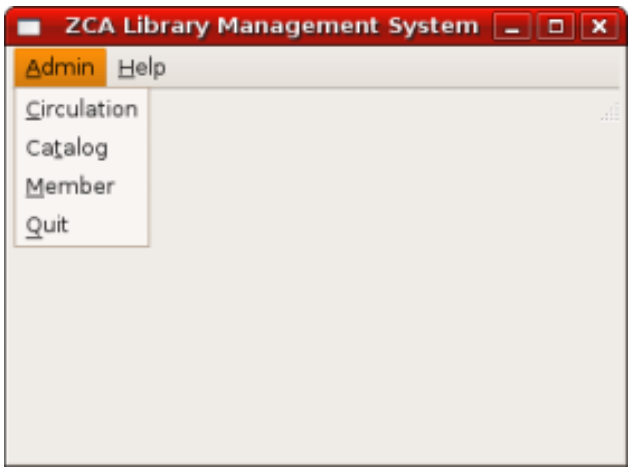
Lorsque l'application grossit, vous pouvez utiliser les composants de la ZCA aux endroits où vous souhaitez fournir de la modularité ou de l'extensibilité. Utilisez des objets Python classiques quand vous n'avez pas besoin de ces propriétés.

Il n'y a pas de différence entre l'utilisation de la ZCA pour une application web, pour une application graphique lourde, ou pour toute autre application ou framework. Il est préférable de suivre une convention pour l'emplacement où vous allez inscrire les composants. Cette application utilise une convention qui peut être étendue en plaçant des inscriptions ou des composants similaires dans des modules séparés et en les important plus tard depuis le module principal d'inscription. Dans cette application, le module principal pour l'inscription est *register.py*.

Le code source de cette application peut être téléchargé sur : http ://www.muthukadan.net/downloads/zcalib.tar.bz2

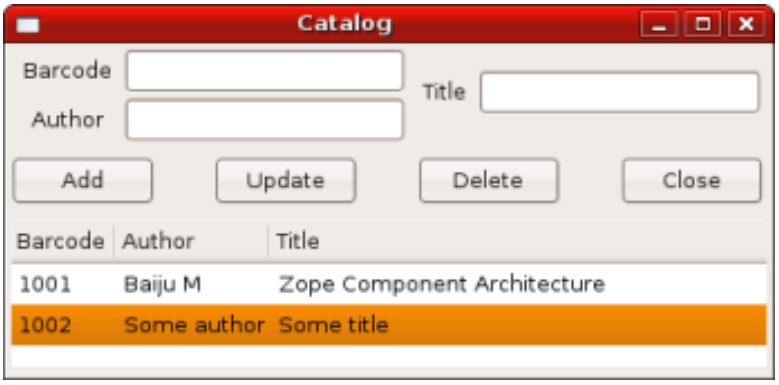
8.2 Cas d'utilisation

L'application que nous allons créer ici est un système de gestion de bibliothèque avec des fonctionnalités minimales. Les besoins peuvent être résumés comme ceci :
– Ajouter des membres avec un numéro et un nom uniques.
– Ajouter des livres avec un code barre, un auteur et un titre.
– Prêter des livres
– Récupérer des livres
L'application peut être conçue de telle façon que les grandes fonctionnalités seront utilisées dans une fenêtre unique. La fenêtre principale peut être conçue de cette façon :

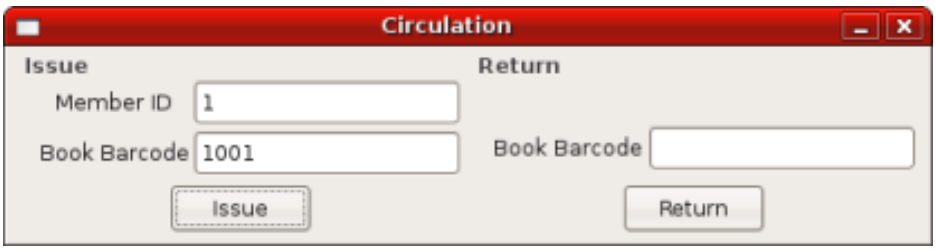


Depuis la fenêtre des membres, l'utilisateur devrait être capable de gérer des membres. Donc il devrait être possible d'*ajouter*, *modifier* et *effacer* des membres comme dans l'image ci-dessous :

Similaire à la fenêtre des membres, la fenêtre du catalogue permet aux utilisateurs d'*ajouter*, *modifier* et *effacer* des livres :



La fenêtre de circulation doit avoir ce qu'il faut pour prêter et récupérer des livres :



8.3 Survol du code PyGTK

Vous pouvez constater que la majorité du code est en rapport avec PyGTK. Sa structure est très similaire pour les différentes fenêtres. Les écrans de cette application sont conçus avec le créateur d'interface graphiques Glade. Vous devriez donner des noms appropriés aux widgets que vous allez utiliser dans votre code. Dans la fenêtre principale, tous les éléments de menu ont des noms du type : circulation, catalog, member, quit et about.

La classe `gtk.glade.XML` est utilisée pour analyser le fichier Glade, et donc pour créer les objets widgets de l'interface graphique. Voici comment faire

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

Dans mainwindow.py, vous pouvez voir le code suivant

```
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')
```

Le nom du widget fenêtre principal est *mainwindow*. Les autres widgets sont créés de la même manière

```
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')
```

Ensuite ces widgets sont connectés à certains événements

```
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)
```

L'événement *delete_event* est celui correspondant à la fermeture de la fenêtre lors de l'appui sur le bouton de fermeture. L'événement *activate* est émis lorsque le menu est sélectionné. Les widgets sont connectés à des fonctions de rappel pour certains événements.

Vous pouvez voir dans le code ci-dessus que la fenêtre principale est connectée à la méthode *on_delete_event* pour l'événement *delete_event*. Le widget *quit* est aussi connecté aux mêmes méthodes pour l'événement *activate*

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

La fonction de rappel lance juste la fonction *main_quit*.

8.4 Le code

Voici *zcalib.py*

```
import registry
import mainwindow

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

Ici, deux modules sont importés : *registry* et *mainwindow*. Ensuite le registre est initialisé et la fonction *main* de la fenêtre principale est appelée. Si l'utilisateur essaye de fermer l'application en appuyant sur *Ctrl-C*, celle-ci s'arrête normalement car nous avons intercepté l'exception *KeyboardInterrupt*.

Voici *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()
```

Observez la fonction *initialize* que nous appelons depuis le module principal *zcalib.py*. Cette fonction commence par vérifier quelle base de données utiliser : une base relationnelle (RDB) ou une base objet (ODB). Cette vérification est effectuée dans la fonction *check_use_relational_db*. Si l'option *-r* est passé dans la ligne de commandes, elle appelera la fonction *initialize_rdb*, sinon *initialize_odb*. Si la fonction RDB est appelée, elle configure tous les composants liés à la base relationnelle. Si la fonction ODB est appelée, elle configure tous les composants liés à la base objet.

Voici *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 fonction *main* crée une instance de la classe *MainWindow* qui initialise tous les widgets.

Voici *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()
```

Voici *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()
```

Voici *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
```

Here is the *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

Chapitre 9

Reference

9.1 adaptedBy

Cette fonction permet de trouver les interfaces adaptées.

- Emplacement : `zope.component`
- Signature : *adaptedBy(object)*

Exemple

```
>>> 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

Un adaptateur peut être n'importe quel objet appelable. Vous pouvez utiliser le décorateur *adapter* pour déclarer qu'un objet appelable s'adapte à des interfaces (ou des classes).

- Emplacement : `zope.component`
- Signature : *adapter(*interfaces)*

Exemple

```
>>> 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) #doctest : +ELLIPSIS
<Job object at ...>
```

9.3 adapts

Cette fonction permet de déclarer les classes adaptateurs.

- Emplacement : `zope.component`
- Signature : *adapts(*interfaces)*

Exemple

```
>>> 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

Déclare des interfaces additionnelles fournies par un objet. Les arguments après l'objet sont une ou plusieurs interfaces. Les interfaces données sont ajoutées aux interfaces précédemment déclarées pour l'objet.

- Emplacement : `zope.interface`
- Signature : *alsoProvides(object, *interfaces)*

Exemple

```
>>> 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)

Vous pouvez tester de cette façon :

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

9.5 Attribute

En utilisant cette classe, vous pouvez définir des attributs dans une interface.

- Emplacement : `zope.interface`
- Signature : *Attribute(name, doc='')*
- Voir aussi : Interface

Exemple

```
>>> 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

Déclare des interfaces additionnelles qui doivent être fournies par les instances d'une classe. Les arguments après la classe sont une ou plusieurs interfaces. Les interfaces données sont ajoutées aux interfaces précédemment déclarées.

- Emplacement : `zope.interface`
- Signature : *classImplements(cls, *interfaces)*

Exemple

```
>>> 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"

Vous pouvez tester de cette façon :

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

9.7 classImplementsOnly

Déclare les seules interfaces qui devront être fournies par les instances d'une classe. Les arguments après la classe sont une ou plusieurs interfaces. Les interfaces fournies vont remplacer toutes les interfaces des déclarations

précédentes.

- Emplacement : `zope.interface`
- Signature : *classImplementsOnly(cls, *interfaces)*

Exemple

```
>>> 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"

Vous pouvez tester de cette façon :

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

9.8 classProvides

Normalement si une classe implémente une interface particulière, les instances de cette classe fourniront l'interface implémentée par la classe. Mais si vous voulez qu'une classe fournisse une interface, vous pouvez le déclarer grâce à la fonction `classProvides`.

- Emplacement : `zope.interface`
- Signature : *classProvides(*interfaces)*

Exemple

```
>>> 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"
```

Vous pouvez tester de cette façon :

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

9.9 ComponentLookupError

C'est l'exception levée quand une recherche de composant échoue.

Exemple

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

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

9.10 createObject

Crée un objet en utilisant une fabrique.

Cette fonction trouve une fabrique nommée dans le site courant et l'appelle avec les arguments donnés. Si la bonne fabrique ne peut pas être trouvée, une erreur `ComponentLookupError` est déclenchée. Sinon l'objet créé est renvoyé.

Un argument mot-clé contextuel peut être fourni pour rechercher une fabrique dans un emplacement différent du site courant. (Bien sûr, cela signifie qu'il est impossible de transmettre à la fabrique un argument mot-clé nommé « context ».

- Emplacement : `zope.component`
- Signature : *createObject(factory_name, *args, **kwargs)*

Exemple

```
>>> 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') #doctest : +ELLIPSIS
<FakeDb object at ...>
```

9.11 Declaration

N'a pas besoin d'être utilisé directement.

9.12 directlyProvidedBy

Cette fonction renvoie les interfaces fournies directement par un objet donné.

- Emplacement : `zope.interface`
- Signature : *directlyProvidedBy(object)*

Exemple

```
>>> 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

Déclare que des interfaces sont fournies directement par un objet. Les arguments après l'objet sont une ou plusieurs interfaces. Les interfaces données remplacent celles précédemment déclarées pour l'objet.

- Emplacement : `zope.interface`
- Signature : *directlyProvides(object, *interfaces)*

Exemple

```
>>> 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

Cette fonction récupère un adaptateur (nommé) vers une interface, pour un objet donné. Elle renvoie un adaptateur qui peut adapter les objets à l'interface voulue. Si aucun adaptateur ne peut être trouvé, une erreur `ComponentLookupError` est émise.

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

Exemple

```
>>> 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') #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
```

9.15 getAdapterInContext

Au lieu d'utiliser cette fonction, utilisez getAdapter avec un argument *context*.

- Emplacement : `zope.component`
- Signature : *getAdapterInContext(object, interface, context)*
- Voir aussi : queryAdapterInContext

Exemple

```
>>> 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) #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
```

9.16 getAdapters

Recherche pour des objets tous les adaptateurs fournissant une interface donnée. Une liste d'adaptateurs est renvoyée. Si un adaptateur est nommé, seul l'adaptateur le plus spécifique pour un nom donné est renvoyé.

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

Exemple

```
>>> 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)) #doctest : +ELLIPSIS
[(u'ng', <FrontDeskNG object at ...>)]
```

9.17 getAllUtilitiesRegisteredFor

Renvoie tous les utilitaires inscrits pour une interface. Ceci inclut les utilitaires surchargés. La valeur renvoyée est un objet iterable listant les instances d'utilitaires correspondants.

- Emplacement : `zope.component`
- Signature : *getAllUtilitiesRegisteredFor(interface)*

Exemple

```
>>> 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) #doctest : +ELLIPSIS
[<Greeter object at ...>]
```

9.18 getFactoriesFor

Renvoie un tuple (nom, fabrique) de fabriques inscrites capables de créer des objets qui fournissent une interface donnée.

- Emplacement : `zope.component`
- Signature : *getFactoriesFor(interface, context=None)*

Exemple

```
>>> 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

Récupère les interfaces implémentées par une fabrique. Cette fonction trouve la fabrique possédant le nom donné la plus proche du contexte, puis renvoie l'interface ou le tuple d'interfaces que les instances créées par la fabrique vont fournir.

- Emplacement : `zope.component`
- Signature : *getFactoryInterfaces(name, context=None)*

Exemple

```
>>> 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

Renvoie le gestionnaire global de site. Cette fonction ne doit jamais échouer et doit toujours renvoyer un objet qui fournit *IGlobalSiteManager*.

- Emplacement : `zope.component`
- Signature : *getGlobalSiteManager()*

Exemple

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

9.21 getMultiAdapter

Cette fonction recherche pour des objets un multi-adaptateur vers une interface donnée. Elle renvoie un multi-adaptateur qui peut s'adapter aux objets donnés et fournir l'interface voulue. Si aucun adaptateur ne peut être trouvé, une erreur *ComponentLookupError* est émise. Le nom constitué de la chaîne vide est réservé aux adaptateur sans nom. Les méthodes des adaptateurs sans nom appellent souvent les méthodes des adaptateurs nommés en fournissant un nom vide (").

- Emplacement : `zope.component`
- Signature : *getMultiAdapter(objects, interface=Interface, name=", context=None)*
- Voir aussi : queryMultiAdapter

Exemple

```
>>> 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) #doctest : +ELLIPSIS
<MyFunctionality object at ...>

>>> myfunctionality = getMultiAdapter((one,two), IFunctionality)
>>> myfunctionality.one #doctest : +ELLIPSIS
<One object at ...>
>>> myfunctionality.two #doctest : +ELLIPSIS
<Two object at ...>
```

9.22 getSiteManager

Récupère le gestionnaire de site le plus proche du contexte donné. Si *context* est *None*, cette fonction renvoie le gestionnaire global de site. Si le *context* n'est pas *None*, on s'attend à ce qu'un adaptateur sur le contexte vers *IComponentLookup* soit trouvé. Si aucun adaptateur n'est trouvé, une erreur *ComponentLookupError* est émise.

- Emplacement : `zope.component`
- Signature : *getSiteManager(context=None)*

Exemple 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
```

Exemple 2

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

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

9.23 getUtilitiesFor

Récupère les utilitaires inscrits pour une interface particulière. Renvoie un objet iterable listant les paires nom/utilitaire.

- Emplacement : `zope.component`
- Signature : *getUtilitiesFor(interface)*

Exemple

```
>>> 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)) #doctest : +ELLIPSIS
[(u'', <Greeter object at ...>)]
```

9.24 getUtility

Récupère l'utilitaire qui fournit une interface donnée. Cette fonction renvoie l'utilitaire le plus proche du contexte et qui fournit l'interface donnée. Si aucun n'est trouvé, une erreur `ComponentLookupError` est levée.

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

Exemple

```
>>> 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

Appelle tous les gestionnaires pour un objet donné. Les gestionnaires sont des fabriques d'abonnés qui ne produisent rien. Ils font tout leur travail lors de l'appel. Ils sont typiquement utilisés pour gérer des événements.

- Emplacement : `zope.component`
- Signature : *handle(*objects)*

Exemple

```
>>> 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

Renvoie les interfaces implémentées par une classe.

- Emplacement : `zope.interface`
- Signature : *implementedBy(class_)*

Exemple 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>
```

Exemple 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

Pour récupérer une liste de toutes les interfaces implémentées par cette
classe : :

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

9.27 implementer

Créer un décorateur permettant de déclarer des interfaces implémentées par une fabrique. Un objet appelable est renvoyé, qui fait une déclaration d'implémentation sur les objets qui lui sont transmis.

- Emplacement : `zope.interface`
- Signature : *implementer(*interfaces)*

Exemple

```
>>> 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

Déclare que des interfaces seront fournies par une classe. Cette fonction est appelée dans une définition de classe. Les arguments sont une ou plusieurs interfaces. Les interfaces données sont ajoutées aux interfaces précédemment déclarées. Les déclarations précédentes incluent les déclarations des classes debase, à moins que *implementsOnly* ne soit utilisé.

- Emplacement : `zope.interface`
- Signature : *implements(*interfaces)*

Exemple

```
>>> 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"

You can test it like this :

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

9.29 implementsOnly

Déclare les seules interfaces qui doivent être implémentées par une classe. Cette fonction est appelée dans la définition de classe. Les arguments sont une ou plusieurs interfaces. Les déclarations précédentes, y compris celles des classes de base, sont remplacées.

- Emplacement : `zope.interface`
- Signature : *implementsOnly(*interfaces)*

Exemple

```
>>> 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"

You can test it like this :

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

9.30 Interface

Grâce à cette classe, vous pouvez définir une interface. Pour définir une interface, contentez-vous d'hériter de cette classe.

- Emplacement : `zope.interface`
- Signature : *Interface(name, doc=")*

Exemple 1

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

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

Exemple 2

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

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

9.31 moduleProvides

Déclare les interfaces fournies par un module. Cette fonction est utilisée dans une définition de module. Les arguments sont une ou plusieurs interfaces. Les interfaces fournies sont utilisées pour créer la spécification d'interface objet du module. Une erreur est levée si le module a déjà une spécification d'interface. Autrement dit, c'est une erreur d'appeler cette fonction plus d'une fois dans une définition de module.

Cette fonction est fournie pour des raisons pratiques. Elle fournit une manière plus pratique d'appeler `directlyProvides` pour un module.

- Emplacement : `zope.interface`
- Signature : *moduleProvides(*interfaces)*
- Voir aussi : directlyProvides

Vous pouvez consulter un exemple d'utilisation dans le code source de *zope.component* lui-même. Le fichier *__init__.py* possède celle instruction

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

Ainsi, le paquet *zope.component* fournit deux interfaces : *IComponentArchitecture* et *IComponentRegistrationConvenience*.

9.32 noLongerProvides

Retire une interface de la liste des interfaces directements fournies par un objet.

- Emplacement : `zope.interface`
- Signature : *noLongerProvides(object, interface)*

Exemple

```
>>> 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)

You can test it like this :

>>> 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

Il est recommandé d'utilier registerAdapter .

9.34 provideHandler

Il est recommandé d'utiliser registerHandler .

9.35 provideSubscriptionAdapter

Il est recommandé d'utiliser registerSubscriptionAdapter .

9.36 provideUtility

Il est recommandé d'utiliser registerUtility .

9.37 providedBy

Teste si l'interface est implémentée par l'objet. Renvoie True si l'objet affirme qu'il implémente l'interface, y compris les interfaces étendues.

- Emplacement : `zope.interface`
- Signature : *providedBy(object)*

Exemple 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"

You can test it like this :

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

Exemple 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"

Pour obtenir une liste de toutes les interfaces fournies par cet objet : :

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

9.38 queryAdapter

Recherche pour un objet un adaptateur nommé vers une interface. Renvoie un adaptateur qui peut s'adapter à un objet et fournir une interface. Si aucun adaptateur ne peut être trouvé, la valeur par défaut est renvoyée.

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

Exemple

```
>>> 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') #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
```

9.39 queryAdapterInContext

Recherche pour un objet un adaptateur spécial vers une interface.

NOTE : cette méthode ne doit être utilisée que si un contexte personnalisé doit être fourni lors d'une recherche de composant personnalisée. Sinon, appelez l'interface comme ci-dessous

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

Renvoie un adaptateur qui peut s'adapter à l'objet et fournir l'interface. Si aucun adaptateur n'est fourni, la valeur par défaut est renvoyée.

Le contexte est adapté à IServiceService, et ce service d'adaptateurs, fourni par l'adaptateur, est utilisé.

Si l'objet possède une méthode __conform__, cette méthode est appelée avec l'interface demandée. Si la méthode renvoie une valeur différente de None, cette valeur est retournée. Sinon, si l'objet implémente déjà l'interface, cet objet est renvoyé.

- Emplacement : `zope.component`
- Signature : *queryAdapterInContext(object, interface, context, default=None)*
- Voir aussi : getAdapterInContext

Exemple

```
>>> 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) #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
```

9.40 queryMultiAdapter

Recherche pour des objets un multi-adaptateur vers une interface. Cette fonction renvoie un multi-adaptateur qui peut adapter les objets à l'interface. Si aucun adaptateur ne peut être trouvé, la valeur par défaut est renvoyée. Le nom correspondant à la chaîne vide est réservé aux adaptateurs sans nom. Les methodes des adaptateurs sans nom appellent souvent les méthodes des adaptateurs nommés avec un nom vide (").

- Emplacement : `zope.component`
- Signature : *queryMultiAdapter(objects, interface=Interface, name=u", default=None, context=None)*
- Voir aussi : getMultiAdapter

Exemple

```
>>> 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) #doctest : +ELLIPSIS
<MyFunctionality object at ...>

>>> myfunctionality = queryMultiAdapter((one,two), IFunctionality)
>>> myfunctionality.one #doctest : +ELLIPSIS
<One object at ...>
>>> myfunctionality.two #doctest : +ELLIPSIS
<Two object at ...>
```

9.41 queryUtility

Cette fonction est utilisée pour rechercher un utilitaire qui fournit une interface. Si aucun n'est trouvé, la valeur par défaut est renvoyée.

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

Exemple

```
>>> 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

Cette fonction est utilisée pour inscrire une fabrique d'adaptateur.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registerAdapter(factory, required=None, provided=None, name=u'', info=u'')*
- Voir aussi : unregisterAdapter

Exemple

```
>>> 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')

Vous pouvez tester de cette façon :

>>> queryAdapter(jack, IDesk, 'ng') #doctest : +ELLIPSIS
<FrontDeskNG object at ...>
```

9.43 registeredAdapters

Renvoie un objet iterable listant les *IAdapterRegistrations*. Ces inscriptions décrivent les inscriptions d'adaptateurs actuelles valables.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registeredAdapters()*

Exemple

```
>>> 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

Renvoie un objet iterable listant les *IHandlerRegistrations*. Ces inscriptions décrivent les inscriptions de gestionnaires actuelles.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registeredHandlers()*

Exemple

```
>>> 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

Renvoie un objet iterable listant les *ISubscriptionAdapterRegistrations*. Ces inscriptions décrivent les inscriptions d'adaptateurs d'abonnement actuelles.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registeredSubscriptionAdapters()*

Exemple

```
>>> 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

Cette fonction renvoie un objet iterable listant les *IUtilityRegistrations*. Ces inscriptions décrivent les inscriptions d'utilitaires actuelles.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registeredUtilities()*

Exemple

```
>>> 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

Cette fonction est utilisée pour inscrire un gestionnaire. Un gestionnaire est un abonné qui ne crée pas d'adaptateur, mais effectue un traitement lorsqu'il est appelé.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registerHandler(handler, required=None, name=u'', info='')*
- Voir aussi : unregisterHandler

Note : l'implémentation actuelle de `zope.component` ne prend pas en charge l'attribut *name*.

Exemple

```
>>> 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

Cette fonction est utilisée pour inscrire une fabrique d'abonnés.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registerSubscriptionAdapter(factory, required=None, provides=None, name=u'', info='')*
- Voir aussi : unregisterSubscriptionAdapter

Exemple

```
>>> 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

Cette fonction est utilisée pour inscrire un utilitaire.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *registerUtility(component, provided=None, name=u'', info=u'')*
- Voir aussi : unregisterUtility

Exemple

```
>>> 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

Cette fonction est utilisée pour récupérer les abonnés. Les abonnés qui fournissent l'interface souhaités et qui dépendent de la séquence d'objets fournie sont renvoyés.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *subscribers(required, provided, context=None)*

Exemple

```
>>> 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

Cette fonction est utilisée pour désinscrire une fabrique d'adaptateur. Un booléen est renvoyé, indiquant si le registre a été modifié. Si le composant donné est None et qu'il n'y a aucun composant inscrit, ou si le composant donné n'est pas None mais n'est pas inscrit, alors la fonction renvoie False. Sinon True.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *unregisterAdapter(factory=None, required=None, provided=None, name=u")*
- Voir aussi : registerAdapter

Exemple

```
>>> 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')

Vous pouvez tester de cette façon :

>>> queryAdapter(jack, IDesk, 'ng6') #doctest : +ELLIPSIS
<FrontDeskNG object at ...>

Maintenant on peut faire la désinscriptin :

>>> gsm.unregisterAdapter(FrontDeskNG, name='ng6')
True

Après la désinscription :

>>> print queryAdapter(jack, IDesk, 'ng6')
None
```

9.52 unregisterHandler

Cette fonction est utilisée pour désinscrire un gestionnaire. Un gestionnaire est un abonné qui ne crée pas d'adaptateur, mais effectue un traitement lors de son appel. Un booléen et renvoyé, indiquant si le registre a été modifié.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *unregisterHandler(handler=None, required=None, name=u")*
- Voir aussi : registerHandler

Exemple

```
>>> 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

Now unregister :

>>> gsm.unregisterHandler(documentAccessed)
True

After unregistration :

>>> handle(DocumentAccessed(doc))
>>> doc.count
0
```

9.53 unregisterSubscriptionAdapter

Cette fonction est utilisée pour désinscrire une fabrique d'abonné. Un booléen est renvoyé, indiquant si le registre a été modifié. Si le composant donné est None et qu'il n'y a aucun composant inscrit, ou si le composant donné n'est pas None mais n'est pas inscrit, alors la fonction renvoie False. Sinon True.

- Emplacement : `zope.component - IComponentRegistry`
- Signature : *unregisterSubscriptionAdapter(factory=None, required=None, provides=None, name=u")*

– Voir aussi : registerSubscriptionAdapter

Exemple

```
>>> 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']
```

```
Mainenant on peut effectuer la désinscription :

>>> gsm.unregisterSubscriptionAdapter(AdequateLength)
True

Après la désinscription :

>>> [adapter.validate()
...  for adapter in subscribers([doc], IValidate)
...  if adapter.validate()]
[]
```

9.54 unregisterUtility

Cette fonction est utilisée pour désinscrire un utilitaire. Un booléen est renvoyé, indiquant si le registre a été modifié. Si le composant donné est None et qu'il n'y a aucun composant inscrit, ou si le composant donné n'est pas None mais n'est pas inscrit, alors la fonction renvoie False. Sinon True.

- Location : `zope.component - IComponentRegistry`
- Signature : *unregisterUtility(component=None, provided=None, name=u")*
- See also : registerUtility

Exemple

```
>>> 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'

Maintenant on peut effectuer la désinscription :

>>> gsm.unregisterUtility(greet)
True
```

Après la désinscription :

```
>>> print queryUtility(IGreeter)
None
```