Zope Component Architecture

Auteur: Baiju M
Version: 0.4.1
URL: http://www.muthukadan.net/docs/zca.pdf
Traducteur: Stéphane Klein <stephane@harobed.org>

Note

Merci à Kent Tenney (Wisconsin, USA) et Brad Allen (Dallas, USA) pour leurs suggestions.

Contents

Chapter 1

Premiers pas

1.1 Introduction

Le développement de logiciels de grandes tailles (grande quantité de lignes de code...) est toujours une tâche très complexe. L'expérience montre que l'utilisation d'une approche orienté objet simplifie l'analyse, la modélisation et l'implémentation de programmes complexes. Les logiciels informatiques basés sur une architecture et une programmation oriénté composant sont de nos jours de plus en plus populaire. Une grande partie des langages de programmations disposent de framework qui permettent ou simplifient la programmation orienté composant. Certain de ces frameworks sont même compatibles avec plusieurs langages. Parmi ces frameworks orientés composants on trouve par exemple la technologie COM de Microsoft ou alors la technologie XPCOM de Mozilla.

L'architecture composant de Zope (nommé ZCA : Zope Component Architecture) est un framework qui permet d'écrire des programmes orientés composants en langage Python. ZCA est très bien adapté pour écrire des programmes Python complexes. Ce framework n'est pas limité au serveur web d'application Zope, il peut être utilisé pour l'écriture de tout type d'application Python, peut être même qu'il devrait être nommé *Python Component Architecture* plutôt que Zope Component Architecture.

ZCA dépend principalement de deux paquets:

- `zope.interface` utilisé pour définir l'interface des composants.
- `zope.component` pour réaliser les opérations d'enregistrements et de recherches de composants.

ZCA dispose de tout ce qu'il faut pour utiliser efficacement des objets Python. Les composants sont des objets réutilisables disposant d'interface est introspectable. Un composant est un objet disposant d'une interface implémenté par une classe ou tout autre objet exécutable. L'implémentation d'un composant n'est pas l'élément le plus important, ce qui compte avant tout c'est sa conformité avec le "contrat" défini par son interface. L'utilisation d'une architecture orienté composant vous permet de diviser la compléxité d'un système à travers de multiples composants coopératifs. L'architecture composant de Zope vous aide à créer deux types de composants de base : des *adapter* et des *utility*.

Notez bien que l'architecture composant de Zope ne vous aide pas à écrire les composants eux-mêmes mais elle vous permet la création, l'enregistrement et la recherche de composants.Notez aussi qu'un composant *adapter* est une classe Python tout à fait normal (ou en général une fabrique d'objet (factory)) et qu'un composant *utility* est un objet Python exécutable standard.

L'architecture composant Zope est développé comme une partie distincte du projet Zope 3. ZCA, comme indiqué plus tôt, est un framework purement Python, il peut être utilisé pour développer n'importe quel type d'application. Actuellement à la fois Zope 3 et Zope 2 utilisent très fortement ce framework mais il existe de nombreux autres projets et même des applications non orientés Internet qui utilisent l'architecture composant de Zope[1].

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 aquise durant le développement d'application d'envergure en Zope 2. Jim Fulton est le père de ce projet. De nombreuses personnes ont participé à la conception de ce framework, parmit eux on peut citer : Stephan Richter, Philipp von Weitershausen, Guido van Rossum (créateur du langage Python), Tres Seaver, Phillip J Eby et Martijn Faassen.

Par rapport à la version actuel, les premières versions de ZCA disposaient de quelques composants supplémentaires : les composants *services* et les composants *views*. Ils ont été supprimé car les développeurs ont réalisé que le composant *utility* pouvait remplacer le composant *service* et le composant *multi-adapter* pouvait remplacer le composant *view*. La ZCA comporte actuellement un nombre réduit de type de type de composant :les *utility*, les *adapter*, les *subscribers* et les *handler*. En fait, les composants *subscriber* et *handler* sont tous deux des adaptateurs spécialisés.

Pendant le cycle de développement de Zope 3.2, Jim Fulton a proposé une importante simplification de l'achitecture composant de Zope[2]. Parmit ces simplications, on trouve la possibilité en passant par une interface unique de déclarer à la fois des composants globaux et locaux.

Le paquet `zope.component` a une longue liste de dépendances, parmit elles beaucoup ne sont pas nécessaire aux applications non Zope 3. Pendant le PyCon 2007, Jim Fulton a ajouté à setuptools une nouvelle fonctionnalité nommée *extras_require*. Elle permet de séparer les fonctionnalités constituant le coeur de ZCA des fonctionnalités annexes[3].

Maintenant, le projet Zope Component Architecture est un projet indépendant avec son propre cycle de version et son propre dépôt Subversion. Toutefois, les bugs et problèmes sont encore traités à l'intérieur du projet Zope 3[4], et la liste de diffusion zope-dev est utilisé pour les discussions liées aux développements[5].

1.3 Installation

Les paquets `zope.component` et `zope.interface` forment ensemble le coeur de l'architecture composant de Zope. Ils fournissent les fonctionnalités permettant de définir, déclarer et retrouver des composants. Le paquet `zope.component` et ses dépendances sont disponibles sous forme de PythonEggs dans le dépôt PyPI (Python Package Index)[6].

Vous pouvez utiliser *easy_install*[7] pour installer `zope.component` et ses dépendances :

```
$ easy_install zope.component
```

Cette commande télécharge `zope.component` et ses depandances depuis le dépôt PyPI et les installe sur votre système.

D'une manière alternative, vous pouvez télécharger manuellement le paquet `zope.component` et ses dépendances depuis PyPi puis les installer manuellement. L'ordre d'installation des paquets vous est donné ci-dessous. Sous Windows, vous aurais besoin du paquet binaire de `zope.interface` et `zope.proxy`.

1. `zope.interface`
2. `zope.proxy`
3. `zope.deferredimport`
4. `zope.event`
5. `zope.deprecation`
6. `zope.component`

Pour installer ces paquets après les avoir téléchagré, vous pouvez utiliser la commande `easy_install` avec le nom des paquets comme argument (vous pouvez aussi donner la totalité des noms des paquets en argument sur la même ligne):

```
$ easy_install /path/to/zope.interface-3.4.x.tar.gz
$ easy_install /path/to/zope.proxy-3.4.x.tar.gz
...
```

Cette méthode install le framework ZCA dans votre *system Python*, dans le répertoire `site-packages`... ce qui peut poser certain soucis. Dans un poste présent dans la liste de diffusion Zope 3, Jim Fulton est défavorable à l'utilisation du *system Python*[8] (c'est à dire à l'installation dans `site-packages`).

1.4 Environement d'expérimentation

`virtualenv` et `zc.buildout` sont des outils qui permettent l'installation de la ZCA dans un environement isolé. L'utilisation de ces outils est conseillé lors l'expérimentation de code. De plus, il est bénéfique de se familiariser avec eux car ils sont très pratiques autant lors de la phase de développement que lors du déploiement d'applications.

Vous pouvez installer"virtualenv" via `easy_install`:

```
$ easy_install virtualenv
```

Vous pouvez créez ensuite un nouvel environement comme ceci:

```
$ virtualenv myve
```

Cette command a pour effet de créer un nouvel environement virtuel dans le dossier `myve`. Maintenant, à partir du dossier `myve`, vous pouvez installer `zope.component` et ses dépendances en utilisant l'exécutable `easy_install` qui se trouve dans le dossier `myve/bin`:

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

Vous pouvez maintenant importer les modules `zope.interface` et `zope.component` a partir de votre nouvel interpréteur `python` qui se trouve dans le dossier `myve/bin`:

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

Cette commande vous donnera une invite de commande Python que vous pouvez utiliser pour exécuter les codes sources de ce livre.

L'utilisation de la recette `zc.recipe.egg` du paquet `zc.buildout` vous permet de créer un interpréteur python avec des PythonEggs spécifique. Premièrement, installez `zc.buildout` en utilisant la commande `easy_install` (Vous pouvez aussi faire cela à l'intérieur d'un environement virtuel). Pour créer un nouveau buildout pour expériement des PythonEggs, commencez par créer un nouveau dossier et initialisez le en utilisant la command `buildout init`:

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

Maintenant, le nouveau dossier `mybuildout` est un buildout. Le fichier de configuration par défaut du buildout est *buildout.cfg*. Après l'initilisation, il a pour contenu:

```
[buildout]
parts =
```

Vous pouvez le modifier comme ceci:

```
[buildout]
parts = py

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

Maintenant exécutez la commande `buildout` disponible dans le dossier `mybuildout/bin` sans aucun argument. Cela a pour effet de créer un nouvel interpréteur à l'intérieur du dossier `mybuildout/bin`:

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

Cette commande vous donnera un invite de commande Python que vous pourrez utiliser pour tester le code source de ce livre.

[1] http://wiki.zope.org/zope3/ComponentArchitecture
[2] http://wiki.zope.org/zope3/LocalComponentManagementSimplification
[3] http://peak.telecommunity.com/DevCenter/setuptools#declaring-dependencies
[4] https://bugs.launchpad.net/zope3
[5] http://mail.zope.org/mailman/listinfo/zope-dev
[6] Repository of Python packages: http://pypi.python.org/pypi
[7] http://peak.telecommunity.com/DevCenter/EasyInstall
[8] http://article.gmane.org/gmane.comp.web.zope.zope3/21045

Chapter 2

An example

2.1 Introduction

Consider a business application for registering guests staying in a hotel. Python can implement this in a number of ways. We will start with a brief look at a procedural implementation, and then move to a basic object oriented approach. As we examine the object oriented approach, we will see how we can benefit from the classic design patterns, *adapter* and *interface*. This will bring us into the world of the Zope Component Architecture.

2.2 Procedural approach

In any business application, data storage is very critical. For simplicity, this example use a Python dictionary as the storage. Key of the dictionary will be the unique Id for a particular guest. And value will be another dictionary with key as the property name:

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

In a simplistic method, a function which accepts details as arguments is enough to do the registration. You also required a supporting function to get the next Id for your data storage.

The supporting function, for getting the next Id can be implemented like this:

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

As you can see, the *get_next_id* function implementation is very simple. Well, this is not the ideal way, but it is sufficient to explain concepts. The function first get all keys of storage as list and check whether it is empty or not. If the list is empty, so no item is stored, it return *1* as the next Id. And if the list is not empty, the next Id is calculated by adding *1* to the maximum value of list.

The function to register guest can get next unique Id using *get_next_id* function, then assign the details of guest using a dictionary. Here is the function to get details and store in the database:

```
>>> def register_guest(name, place):
...     next_id = get_next_id()
...     guests_db[next_id] = {
...     'name': name,
...     'place': place
...     }
```

We will end our discussion of the procedural approach here. It will be much easier to add required features such as data persistence, design flexibility, and code testability using objects.

2.3 Object oriented approach

In object oriented methodology, you can think of a registrar object handling the registration. There are many advantages for creating an object for handling registration. Most importantly, the abstraction provided by the registrar object makes the code easier to understand. It offers a way to group related functionality, and can be extended via inheritance. As features are added, such as canceling and updating registration, the registrar object can grow to provide them, or delegate them to another object.

Lets look at the implementation details of a registrar object implemented as a class:

```
>>> class guestRegistrar(object):
...
...     def register(self, name, place):
...         next_id = get_next_id()
...         guests_db[next_id] = {
...         'name': name,
...         'place': place
...         }
```

In this implementation, the registrar object (an instance of *GuestRegistrar* class) is handling the registration. With this design, a particular registrar object can perform multiple registrations.

This is how you can use the current registrar implementation:

```
>>> registrar = GuestRegistrar()
>>> registrar.register("Jack", "Bangalore")
```

Requirement changes are unavoidable in any real world project. Consider this case, after some time, a new requirement is arising: the guests also required to provide phone number to admit them. You will be required to change the implementation of registrar object to support it.

You can achieve this requirement by adding one argument to *registrar* method and use that argument in the dictionary of values. Here is the new implementation for this requirement:

```
>>> class GuestRegistrar(object):
...
...     def register(self, name, place, phone):
...         next_id = get_next_id()
...         guests_db[next_id] = {
...         'name': name,
...         'place': place,
...         'phone': phone
...         }
```

Other than migrating the data to new schema, now you have to change the usage of *GuestRegistrar* in all places. If you can abstract the details of guest into an object and use it for registration, the code changes can be minimized. If you follow this design, you have to pass that guest object instead of more arguments to the function. The new implementation with guest object will look like this:

```
>>> class GuestRegistrar(object):
...
...     def register(self, guest):
...         next_id = get_next_id()
...         guests_db[next_id] = {
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }
```

Well, even in this implementation you have to change code. Code change for requirement is unavoidable, your goal should be to minimize changes and make it maintainable.

> **Note**
>
> You should have the courage to make any change, major or minor, at any time. Immediate feedback is the only way you can get the courage. Using automated testing, you can get the immediate feedback and so the courage to make changes. For more details about this subject, you can read the book called *Extreme Programming Explained* by Kent Beck.

By introducing the guest object, you saved some typing. More than that, the abstraction of guest object made the system much simpler and easy to understand. The better understanding leads to better restructuring and hence maintainable code.

2.4 The adapter pattern

In a real application, as you noted earlier, the registrar object may have cancellation and/or updation functionalities. Suppose there are two more methods like, *cancel_registration* and *update_registration*. In the new design you will be required to pass the guest object for each methods. You can solve this problem by setting guest object as an attribute of registrar object.

Here is the new implementation of registrar object which set guest object as an attribute:

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

The solution you reached is a common design pattern called, *Adapter*. With this design, now you can add more methods, so more functionality, if required.

In this implementation, while creating the instance you have to pass the guest object which has the values as attributes. Now you also required to create separate instances of *GuestRegistrarNG* for each guest object.

Now just step back and think differently. Suppose you are the creator of this software and selling it to many hotel customers. Consider a case where your different clients requires different storages. For example, one registrar might store the details in a relational database and another one might store them in Zope Object Database (ZODB). It would be better if you can replace the registrar object with another one which store guest details in a different way. So, a mechanism to change implementation based on some configuration would be useful.

Zope component architecture provides a mechanism to replace components based on configuration. Using Zope component architecture you can register components in a registry called component registry. Later, retrieve component based on the configuration.

The *GuestRegistrarNG* class follows, as you noted, a pattern called *Adapter*. The *GuestRegistrarNG* is the adapter which adapts the guest object (adaptee). As you can see, the adapter should contain the component it adapts (adaptee). This is a typical implementation of adapter:

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

Now the adapter can make use adaptee (call its methods or access attributes). An adapter may adapt more than one component. Zope component architecture provides a mechanism to effectively use these kind of objects. So, which component should be used will become a matter of configuration.

This is a common scenario where you want to use different objects doing same things, but the details may change. There are many situations in programming where you want to use different implementations for same type of objects. Here is a small list of other common scenarios:

- A wiki engine with support for multiple markups (STX, reST, Plain text, etc.)
- An object browser which shows size of different types of objects.
- Different types of output formats for text data (PDF, HTML etc.)
- When developing an application for multiple clients, their requirements may change. Maintaining separate code bases of the same application for different clients is difficult. A better approach would be to create reusable components and configure them based on client-specific requirements.

All these examples points to situations where you want to make applications extensible or pluggable. Do not use *adapter* components where you do not want extensibility or pluggability.

Zope component architecture provides *adapter* components to solve these kinds of problems. In fact, *GuestRegistrarNG* is an adapter without explicit interface declaration. This tutorial will discuss adapters after introducing the concept of interfaces. Interfaces are one of the foundations of Zope components, so understanding the concept and usage of interfaces is very important.

Chapter 3

Interfaces

3.1 Introduction

Design Patterns is a classic book in software engineering by the *Gang of Four*[9]. In this book they recommend: "Program to an interface, not an implementation". Defining formal interfaces helps you better understand system. Moreover interface brings you all the benefits of ZCA.

Interface define the behavior and state of objects. An interface describes how you work with the object. If you like metaphor, think of interface as a *contract for object*. Another metaphor which may help is *blueprint for objects*. In the code, methods and attributes are forming the object's interface.

The notion of interface is very explicit in modern languages like Java, C#, VB.NET etc. Also these languages provide some syntax for defining interfaces. Python has the notion interfaces, but it is not very explicit. To simulate a formal definition of interfaces in C++, *Gang of Four* used classes with virtual functions in *Design Patterns* book. In a similar fashion, Zope component architecture use `zope.interface.Interface` inherited meta-class for defining an interface.

The base of object-orientation is the communication between objects. Messages are used for the communication between objects. In Python, functions, methods or any other callable can be used to handle messages.

For example, consider this class:

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

In the above class, you defined a *goodmorning* method. If you call the *goodmorning* method from an object created using this class, it will return *Good morning, ...!*

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

Here `host` is the actual object. The implementation details of this object is the class `Host`. Now, how to find how the object looks like, that is, what are the methods and attributes of the object. For this, either you have go through the implementation details (`Host` class) of the object or a separate API documentation[10] will be required.

You can use the `zope.interface` package to define the interface of objects. For the class given above you can specify the interface like this:

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

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

As you can see, the interface is defined using Python class statement. We use (abuse?) Python's class statement to define interfaces. To make a class an interface, it must be inherited from `zope.interface.Interface` . The `I` prefix for interface name is a convention.

3.2 Declaring interfaces

You have already seen how to declare an interface using `zope.interface` in previous section. This section will explain the concepts in detail.

Consider this example 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"""
```

The interface, `IHost` has two attributes, `name` and `goodmorning`. Recall that, at least in Python, methods are also attributes of classes. The `name` attribute is defined using `zope.interface.Attribute` class. When you add the attribute `name` to the `IHost` interface, you don't set an initial value. The purpose of defining the attribute `name` here is merely to indicate that any implementation of this interface will feature an attribute named `name`. In this case, you don't even say what type of attribute it has to be!. You can pass a documentation string as a first argument to `Attribute`.

The other attribute, `goodmorning` is a method defined using a function definition. Note that *self* is not required in interfaces, because *self* is an implementation detail of class. For example, a module can implement this interface. If a module implement this interface, there will be a `name` attribute and `goodmorning` function defined. And the `goodmorning` function will accept one argument.

Now you will see how to connect *interface-class-object*. So object is the real living thing, objects are instances of classes. And interface is the actual definition of the object, so classes are just the implementation details. This is why you should program to an interface and not to an implementation.

Now you should familiarize two more terms to understand other concepts. First one is *provide* and the other one is *implement*. Object provides interfaces and classes implement interfaces. In other words, objects provide interfaces that their classes implement. In the above example `host` (object) provides `IHost` (interface) and `Host` (class) implement `IHost` (interface). One object can provide more than one interface also one class can implement more than one interface. Objects can also provide interfaces directly, in addition to what their classes implement.

Note

Classes are the implementation details of objects. In Python, classes are callable objects, so why other callable objects can't implement an interface. Yes, it is possible. For any *callable object* you can declare that it produces objects that provide some interfaces by saying that the *callable object* implements the interfaces. The *callable objects* are generally called as *factories*. Since functions are callable objects, a function can be an *implementer* of an interface.

3.3 Implementing interfaces

To declare a class implements a particular interface, use the function `zope.interface.implements` in the class statement.

Consider this example, here `Host` implements `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

If you wonder how `implements` function works, refer the blog post by James Henstridge (http://blogs.gnome.org/jamesh/2005/09/08/python-class-advisors/) . In the adapter section, you will see an `adapts` function, it is also working similarly.

Since `Host` implements `IHost`, instances of `Host` provides `IHost`. There are some utility methods to introspect the declarations. The declaration can write outside the class also. If you don't write `interface.implements(IHost)` in the above example, then after defining the class statement, you can write like this:

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

3.4 Example revisited

Now, return to the example application. Here you will see how to define the interface of the registrar object:

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

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

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

Here, first you imported `Interface` class from `zope.interface` module. If you define a subclass of this `Interface` class, it will be an interface from Zope component architecture point of view. An interface can be implemented, as you already noted, in a class or any other callable object.

The registrar interface defined here is `IRegistrar`. The documentation string for interface gives an idea about the object. By defining a method in the interface, you made a contract for the component, that there will be a method with same name available. For the method definition interface, the first argument should not be *self*, because an interface will never be instantiated nor will its methods ever be called. Instead, the interface class merely documents what methods and attributes should appear in any normal class that claims to implement it, and the *self* parameter is an implementation detail which doesn't need to be documented.

As you know, an interface can also specify normal attributes:

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

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

In this interface, guest object has two attributes specified with documentation. An interface can also specify both attributes and methods together. An interface can be implemented in a class, module or any other objects. For example a function can dynamically create the component and return, in this case the function is an implementer for the interface.

Now you know what is an interface and how to define and use it. In the next chapter you can see how an interface is used to define an adapter component.

3.5 Marker interfaces

An interface can be used to declare that a particular object belongs to a special type. An interface without any attribute or method is called *marker interface*.

Here is a *marker interface*:

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

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

This interface can be used to declare an object is a special guest.

3.6 Invariants

Sometimes you will be required to use some rule for your component which involve one or more normal attributes. These kind of rule is called *invariants*. You can use `zope.interface.invariant` for setting *invariants* for your objects in their interface.

Consider a simple example, there is a *person* object. A person object has *name*, *email* and *phone* attributes. How do you implement a validation rule that says either email or phone have to exist, but not necessarily both.

First you have to make a callable object, either a simple function or callable instance of a class like this:

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

Then define the *person* object's interface like this. Use the `zope.interface.invariant` function to set the 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)
```

Now use *validateInvariants* method of the interface to validate:

```
>>> 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
```

As you can see *jack* object validated without raising any exception. But *jill* object didn't validated the invariant constraint, so it raised exception.

[9] http://en.wikipedia.org/wiki/Design_Patterns
[10] http://en.wikipedia.org/wiki/Application_programming_interface

Chapter 4

Adapters

4.1 Implementation

This section will describe adapters in detail. Zope component architecture, as you noted, helps to effectively use Python objects. Adapter components are one of the basic components used by Zope component architecture for effectively using Python objects. Adapter components are Python objects, but with well defined interface.

To declare a class is an adapter use *adapts* function defined in `zope.component` package. Here is a new *GuestRegistrarNG* adapter with explicit interface declaration:

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

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

What you defined here is an *adapter* for *IRegistrar*, which adapts *IGuest* object. The *IRegistrar* interface is implemented by *GuestRegistrarNG* class. So, an instance of this class will provide *IRegistrar* interface.

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

```
...         self.name = name
...         self.place = place

>>> jack = Guest("Jack", "Bangalore")
>>> jack_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True
```

The *GuestRegistrarNG* is just one adapter you created, you can also create other adapters which handles guest registration differently.

4.2 Registration

To use this adapter component, you have to register this in a component registry also known as site manager. A site manager normally resides in a site. A site and site manager will be more important when developing a Zope 3 application. For now you only required to bother about global site and global site manager (or component registry). A global site manager will be in memory, but a local site manager is persistent.

To register your component, first get the global site manager:

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

To get the global site manager, you have to call `getGlobalSiteManager` function available in `zope.component` package. In fact, the global site manager is available as an attribute (`globalSiteManager`) of `zope.component` package. So, you can directly use `zope.component.globalSiteManager` attribute. To register the adapter in component, as you can see above, use `registerAdapter` method of component registry. The first argument should be your adapter class/factory. The second argument is a tuple of *adaptee* objects, i.e, the object which you are adapting. In this example, you are adapting only *IGuest* object. The third argument is the interface provided by the adapter component. The fourth argument is optional, that is the name of the particular adapter. Since you gave a name for this adapter, this is a *named adapter*. If name is not given, it will default to an empty string (").

In the above registration, you have given the adaptee interface and interface to be provided by the adapter. Since you have already given these details in adapter implementation, it is not required to specify again. In fact, you could have done the registration like this:

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

There are some old API to do the registration, which you should avoid. The old API functions starts with *provide*, eg: `provideAdapter`, `provideUtility` etc. While developing a Zope 3 application you can use Zope configuration markup language (ZCML) for registration of components. In Zope 3, local components (persistent components) can be registered from Zope Management Interface (ZMI) or you can do it programmatically also.

You registered *GuestRegistrarNG* with a name *ng*. Similarly you can register other adapters with different names. If a component is registered without name, it will default to an empty string.

> **Note**
>
> Local components are persistent components but global components are in memory. Global components will be registered based on the configuration of application. Local components are taken to memory from database while starting the application.

4.3 Querying adapter

Retrieving registered components from component registry is achieved through two functions available in `zope.component` package. One of them is `getAdapter` and the other is `queryAdapter` . Both functions accepts same arguments. The `getAdapter` will raise `ComponentLookupError` if component lookup fails on the other hand `queryAdapter` will return *None*.

You can import the methods like this:

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

In the previous section you have registered a component for guest object (adaptee) which provides *IRegistrar* interface with name as 'ng'. In the first section of this chapter, you have created a guest object named *jack* .

This is how you can retrieve a component which adapts the interface of *jack* object (*IGuest*) and provides *IRegistrar* interface also with name as 'ng'. Here both `getAdapter` and `queryAdapter` works similarly:

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

As you can see, the first argument should be adaptee then, the interface which should be provided by component and last the name of adapter component.

If you try to lookup the component with an name not used for registration but for same adaptee and interface, the lookup will fail. Here is how the two methods works in such a case:

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

As you can see above, `getAdapter` raised a `ComponentLookupError` exception, but `queryAdapter` returned *None* when lookup failed.

The third argument, the name of registration, is optional. If the third argument is not given it will default to empty string (''). Since there is no component registered with an empty string, `getAdapter` will raise `ComponentLookupError` . Similarly `queryAdapter` will return *None*, see yourself how it works:

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

In this section you have learned how to register a simple adapter and how to retrieve it from component registry. These kind of adapters is called single adapter, because it adapts only one adaptee. If an adapter adapts more that one adaptee, then it is called multi adapter.

4.4 Retrieving adapter using interface

Adapters can be directly retrieved using interfaces, but it will only work for non-named single adapters. The first argument is the adaptee and the second argument is a keyword argument. If adapter lookup fails, second argument will be returned.

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

Keyword name can be omitted:

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

If second argument is not given, it will raise *TypeError*:

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

Here *GuestRegistrarNG* is registered without name:

```
>>> gsm.registerAdapter(GuestRegistrarNG)
```

Now the adapter lookup should succeed:

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

For simple cases, you may use interface to get adapter components.

4.5 Adapter pattern

The adapter concept in Zope Component Architecture and the classic *adapter pattern* as described in Design Patterns book are very similar. But the intent of ZCA adapter usage is more wider than the *adapter pattern* itself. The intent of *adapter pattern* is to convert the interface of a class into another interface clients expect. This allows classes work together that couldn't otherwise because of incompatible interfaces. But in the *motivation* section of Degisgn Patterns book, GoF says: "Often the adapter is responsible for functionality the adapted class doesn't provide". ZCA adapter has more focus on adding functionalities than creating a new interface for an adapted object (adaptee). ZCA adapter lets adapter classes extend functionality by adding methods. (It would be interesting to note that *Adapter* was known as *Feature* in earlier stage of ZCA design.)[11]

The above paragraph has a quote from Gang of Four book, it ends like this: " ...adapted class doesn't provide". But in the next sentence I used "adapted object" instead of "adapted class", because GoF describes about two variants of adapters based on implementations. The first one is called *class adapter* and the other one is called *object adapter*. A class adapter uses multiple inheritance to adapt one interface to another, on the other hand an object adapter relies on object composition. ZCA adapter is following object adapter pattern, which use delegation as a mechanism for composition. GoF's second principle of object-oriented design goes like this: "Favor object composition over class inheritance". For more details about this subject please read Design Patterns book.

The major attraction of ZCA adapter are the explicit interface for components and the component registry. ZCA adapter components are registered in component registry and looked up by client objects using interface and name when required.

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

Chapter 5

Utility

5.1 Introduction

Now you know the concept of interface, adapter and component registry. Sometimes it would be useful to register an object which is not adapting anything. Database connection, XML parser, object returning unique Ids etc. are examples of these kinds of objects. These kind of components provided by Zope component architecture are called `utility` components.

Utilities are just objects that provide an interface and that are looked up by an interface and a name. This approach creates a global registry by which instances can be registered and accessed by different parts of your application, with no need to pass the instances around as parameters.

You need not to register all component instances like this. Only register components which you want to make replaceable.

5.2 Simple utility

Before implementing the utility, as usual, define its interface. Here is a *greeter* interface:

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

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

Here is a possible implementation of the above interface:

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

You can register an instance of this class using `registerUtility`:

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

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

In this example you registered the utility as providing the *IGreeter* interface. You can look the interface up with either *queryUtility* or *getUtility*:

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

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

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

As you can see, adapters are normally classes, but utilities are normally instances of classes.

5.3 Named utility

When registering a utility component, like adapter, you can use a name.

For example consider this:

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

In this example you registered the utility with a name as providing the *IGreeter* interface. You can look the interface up with either *queryUtility* or *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'
```

As you can see here, while querying you have to use the *name* as second argument.

> **Note**
>
> Calling *getUtility* function without a name (second argument) is equivalent to calling with an empty string (") as the name. Because, the default value for second (keyword) argument is an empty string. Then, component lookup mechanism will try to find the component with name as empty string ("), and it will fail. When component lookup fails it will raise *ComponentLookupError* exception. Remember, it will not return some random component registered with some other name.

5.4 Factory

A `Factory` is a utility component which provides `IFactory` interface.

To create a factory, first define the interface of the object:

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

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

Here is fake implementation of *IDatabase* interface:

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

You can create a factory using `zope.component.factory.Factory`:

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

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

Now you can register it like this:

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

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

To use the factory, you may do it like this:

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

There is a shortcut to use factory:

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

Chapter 6

Advanced adapters

6.1 Multi adapter

A simple adapter normally adapts only one object, but an adapter can adapt more than one object. If an adapter adapts more than one objects, it is called as multi-adapter.

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

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

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

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

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

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

>>> gsm.registerAdapter(MyFunctionality)

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

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

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

>>> from zope.component import getMultiAdapter

>>> getMultiAdapter((one,two), IFunctionality) #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 Subscription adapter

Unlike regular adapters, subscription adapters are used when we want all of the adapters that adapt an object to a particular adapter. Subscription adapter is also known as *subscriber* .

Consider a validation problem. We have objects and we want to assess whether they meet some sort of standards. We define a validation interface:

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

Perhaps we have 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
```

Now, we may want to specify various validation rules for documents. For example, we might require that the summary be a single line:

```
>>> 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 ''
```

Or we might require the body to be at least 1000 characters in length:

```
>>> 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 ''
```

We can register these as subscription adapters:

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

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

We can then use the subscribers to validate objects:

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

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

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

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

6.3 Handler

Handlers are subscription adapter factories that don't produce anything. They do all of their work when called. Handlers are typically used to handle events. Handlers are also known as event subscribers or event subscription adapters.

Event subscribers are different from other subscription adapters in that the caller of event subscribers doesn't expect to interact with them in any direct way. For example, an event publisher doesn't expect to get any return value. Because subscribers don't need to provide an API to their callers, it is more natural to define them with functions, rather than classes. For example, in a document-management system, we might want to record creation times for documents:

```
>>> import datetime

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

In this example, we have a function that takes an event and performs some processing. It doesn't actually return anything. This is a special case of a subscription adapter that adapts an event to nothing. All of the work is done when the adapter "factory" is called. We call subscribers that don't actually create anything "handlers". There are special APIs for registering and calling them.

To register the subscriber above, we define a document-created event:

```
>>> 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
```

We'll also change our handler definition to:

```
>>> 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()
```

This marks the handler as an adapter of *IDocumentCreated* events.

Now we'll register the handler:

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

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

Now, if we can create an event and use the *handle* function to call handlers registered for the event:

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

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

Chapter 7

ZCA usage in Zope

Zope Component Architecture is used in both Zope 3 and Zope 2. This chapter go through ZCA usage in Zope.

7.1 ZCML

The Zope Configuration Markup Language (ZCML) is an XML based configuration system for registration of components. So, instead of using Python API for registration, you can use ZCML. But to use ZCML, unfortunately, you will be required to install more dependency packages.

To install these packages:

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

To register an adapter:

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

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

The *provides* and *for* attributes are optional, provided you have declared it in the implementation:

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

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

If you want to register the component as named adapter, you can give a *name* attribute:

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

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

Utilities are also registered similarly.

To register an utility:

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

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

The *provides* attribute is optional, provided you have declared it in the implementation:

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

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

If you want to register the component as named adapter, you can give a *name* attribute:

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

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

Instead of directly using the component, you can also give a factory:

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

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

7.2 Overrides

When you register components using Python API (`register*` methods), the last registered component will replace previously registered component, if both are registered with same type of arguments. For example, consider this example:

```
>>> 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 ...>

If you register another adapter, the existing one will be replaced:

>>> gsm.registerAdapter(AP2)

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

But when registering components using ZCML, the second registration will raise a conflict error. This is a hint for you, otherwise there is a chance for overriding registration by mistake. This may lead to hard to track bugs in your system. So, using ZCML is a win for the application.

Sometimes you will be required to override existing registration. ZCML provides `includeOverrides` directive for this. Using this, you can write your overrides in a separate file:

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

7.3 NameChooser

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

This is an adapter for choosing a unique name for an object inside a container.

The registration of adapter is like this:

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

From the registration, you can see that the adaptee is a `IWriteContainer` and the adapter provides `INameChooser`.

This adapter provides a very convenient functionality for Zope programmers. The main implementations of `IWriteContainer` in Zope 3 are `zope.app.container.BTreeContainer` and `zope.app.folder.Folder`. Normally you will be inheriting from these implementations for creating your own container classes. Suppose there is no interface called `INameChooser` and adapter, then you will be required to implement this functionality for every implementations separately.

7.4 LocationPhysicallyLocatable

Location: `zope.location.traversing.LocationPhysicallyLocatable`

This adapter is frequently used in Zope 3 applications, but normally it is called through an API in `zope.traversing.api`. (Some old code even use `zope.app.zapi` functions, which is again one more indirection)

The registration of adapter is like this:

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

The interface provided and adaptee interface is given in the implementation.

Here is the beginning of implementation:

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

Normally, almost all persistent objects in Zope 3 application will be providing the `ILocation` interface. This interface has only two attribute, `__parent__` and `__name__`. The `__parent__` is the parent in the location hierarchy. And `__name__` is the name within the parent.

The `IPhysicallyLocatable` interface has four methods: `getRoot`, `getPath`, `getName`, and `getNearestSite`.

- `getRoot` function will return the physical root object.
- `getPath` return the physical path to the object as a string.
- `getName` return the last segment of the physical path.

- `getNearestSite` return the site the object is contained in. If the object is a site, the object itself is returned.

If you learn Zope 3, you can see that these are the important things which you required very often. To understand the beauty of this system, you must see how Zope 2 actually get the physical root object and how it is implemented. There is a method called `getPhysicalRoot` virtually for all container objects.

7.5 DefaultSized

Location: `zope.size.DefaultSized`

This adapter is just a default implementation of `ISized` interface. This adapter is registered for all kind of objects. If you want to register this adapter for a particular interface, then you have to override this registration for your implementation.

The registration of adapter is like this:

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

As you can see, the adaptee interface is *, so it can adapt any kind of objects.

The `ISized` is a simple interface with two method contracts:

```
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.
        """
```

You can see another `ISized` adapter registered for `IZPTPage` in `zope.app.zptpage` package.

7.6 ZopeVersionUtility

Location: `zope.app.applicationcontrol.ZopeVersionUtility`

This utility gives version of the running Zope.

The registration goes like this:

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

The interface provided, `IZopeVersion`, has only one method named `getZopeVersion`. This method return a string containing the Zope version (possibly including SVN information).

The default implementation, `ZopeVersionUtility`, get version info from a file `version.txt` in *zope/app* directory. If Zope is running from subversion checkout, it will show the latest revision number. If none of the above works it will set it to: *Development/Unknown*.

Chapter 8

Reference

8.1 Attribute

Using this class, you can define normal attribute in an interface.

- Location: `zope.interface`
- Signature: *Attribute(name, doc='')*

Example:

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

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

8.2 Declaration

Need not to use directly.

8.3 Interface

Using this class, you can define an interface. To define an interface, just inherit from `Interface` class.

- Location: `zope.interface`
- Signature: *Interface(name, doc='')*

Example 1:

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

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

Example 2:

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

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

8.4 adapts

This function helps to declare adapter classes.

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

Example:

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

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

8.5 alsoProvides

Declare interfaces declared directly for an object. The arguments after the object are one or more interfaces. The interfaces given are added to the interfaces previously declared for the object.

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

Example:

```
>>> 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(IRegistrar)
...     name = u""

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

You can test it like this:

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

8.6 classImplements

Declare additional interfaces implemented for instances of a class. The arguments after the class are one or more interfaces. The interfaces given are added to any interfaces previously declared.

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

Example:

```
>>> 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(IRegistrar)
...     name = u""
...     college = u""

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

You can test it like this:

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

8.7 classImplementsOnly

Declare the only interfaces implemented by instances of a class. The arguments after the class are one or more interfaces. The interfaces given replace any previous declarations.

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

Example:

```
>>> 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"

You can test it like this:

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

8.8 classProvides

Normally if a class implements a particular interface, the instance of that class will provide the interface implemented by that class. But if you want a class to be provided by an interface, you can declare it using `classProvides` function.

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

Example:

```
>>> 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"

You can test it like this:

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

8.9 ComponentLookupError

8.10 createObject

Create an object using a factory.

Finds the named factory in the current site and calls it with the given arguments. If a matching factory cannot be found raises ComponentLookupError. Returns the created object.

A context keyword argument can be provided to cause the factory to be looked up in a location other than the current site. (Of course, this means that it is impossible to pass a keyword argument named "context" to the factory.

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

Example:

```
>>> 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 ...>
```

8.11 directlyProvidedBy

This function will return the interfaces directly provided by the given object.

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

Example:

```
>>> 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
```

8.12 directlyProvides

Declare interfaces declared directly for an object. The arguments after the object are one or more interfaces. The interfaces given replace interfaces previously declared for the object.

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

Example:

```
>>> 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
```

8.13 getAdapter

Get a named adapter to an interface for an object. Returns an adapter that can adapt object to interface. If a matching adapter cannot be found, raises `ComponentLookupError` .

- Location: `zope.interface`

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

Example:

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

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

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

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

>>> getAdapter(jack, IRegistrar, 'ng') #doctest: +ELLIPSIS
```

```
<GuestRegistrarNG object at ...>
```

8.14 getAdapterInContext

Instead of this function, use *context* argument of getAdapter function.

- Location: `zope.component`
- Signature: *getAdapterInContext(object, interface, context)*

Example:

```
>>> 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 IRegistrar(Interface):
...     """A registrar will register object's details"""
...
...     def register():
...         """Register object's details"""
...

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> sm.registerAdapter(GuestRegistrarNG,
...                    (IGuest,), IRegistrar)

>>> from zope.component import getAdapterInContext
>>> from zope.component import queryAdapterInContext

>>> getAdapterInContext(jack, IRegistrar, sm) #doctest: +ELLIPSIS
<GuestRegistrarNG object at ...>
```

8.15 getAdapters

Look for all matching adapters to a provided interface for objects. Return a list of adapters that match. If an adapter is named, only the most specific adapter of a given name is returned.

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

Example:

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_db[next_id] = {
```

```
...         'name': guest.name,
...         'place': guest.place,
...         'phone': guest.phone
...         }

>>> jack = Guest("Jack", "Bangalore")
>>> jack_registrar = GuestRegistrarNG(jack)

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

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

>>> from zope.component import getAdapters
>>> list(getAdapters((jack,), IRegistrar)) #doctest: +ELLIPSIS
[(u'ng', <GuestRegistrarNG object at ...>)]
```

8.16 getAllUtilitiesRegisteredFor

Return all registered utilities for an interface. This includes overridden utilities. The returned value is an iterable of utility instances.

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

Example:

```
>>> 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 ...>]
```

8.17 getFactoriesFor

Return a tuple (name, factory) of registered factories that create objects which implement the given interface.

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

Example:

```
>>> 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'>>)]
```

8.18 getFactoryInterfaces

Get interfaces implemented by a factory. Finds the factory of the given name that is nearest to the context, and returns the interface or interface tuple that object instances created by the named factory will implement.

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

Example:

```
>>> 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>
```

8.19 getGlobalSiteManager

Return the global site manager. This function should never fail and always return an object that provides *IGlobalSiteManager*

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

Example:

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

8.20 getMultiAdapter

Look for a multi-adapter to an interface for an objects. Returns a multi-adapter that can adapt objects to interface. If a matching adapter cannot be found, raises ComponentLookupError. The name consisting of an empty string

is reserved for unnamed adapters. The unnamed adapter methods will often call the named adapter methods with an empty string for a name.

- Location: `zope.component`
- Signature: *getMultiAdapter(objects, interface=Interface, name='', context=None)*

Example:

```
>>> 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 ...>
```

8.21 getSiteManager

Get the nearest site manager in the given context. If *context* is *None*, return the global site manager. If the *context* is not *None*, it is expected that an adapter from the *context* to *IComponentLookup* can be found. If no adapter is found, a *ComponentLookupError* is raised.

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

Example 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
```

Example 2:

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

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

8.22 getUtilitiesFor

Look up the registered utilities that provide an interface. Returns an iterable of name-utility pairs.

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

Example:

```
>>> 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 ...>)]
```

8.23 getUtility

Get the utility that provides interface. Returns the nearest utility to the context that implements the specified interface. If one is not found, raises ComponentLookupError.

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

Example:

```
>>> 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'
```

8.24 handle

Call all of the handlers for the given objects. Handlers are subscription adapter factories that don't produce anything. They do all of their work when called. Handlers are typically used to handle events.

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

Example:

```
>>> 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'
```

8.25 implementedBy

Return the interfaces implemented for a class' instances.

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

Example 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>
```

Example 2:

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

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

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

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

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

```
>>> from zope.interface import implementedBy

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

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

8.26 implementer

Create a decorator for declaring interfaces implemented by a factory. A callable is returned that makes an implements declaration on objects passed to it.

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

Example:

```
>>> 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>]
```

8.27 implements

Declare interfaces implemented by instances of a class This function is called in a class definition. The arguments are one or more interfaces. The interfaces given are added to any interfaces previously declared. Previous declarations include declarations for base classes unless implementsOnly was used.

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

Example:

```
>>> 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
```

8.28 implementsOnly

Declare the only interfaces implemented by instances of a class. This function is called in a class definition. The arguments are one or more interfaces. Previous declarations including declarations for base classes are overridden.

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

Example:

```
>>> 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
```

8.29 moduleProvides

Declare interfaces provided by a module. This function is used in a module definition. The arguments are one or more interfaces. The given interfaces are used to create the module's direct-object interface specification. An error will be raised if the module already has an interface specification. In other words, it is an error to call this function more than once in a module definition.

This function is provided for convenience. It provides a more convenient way to call directlyProvides for a module.

- Location: `zope.interface`
- Signature: *moduleProvides(*interfaces)*

8.30 noLongerProvides

Remove an interface from the list of an object's directly provided interfaces.

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

Example:

```
>>> 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
```

8.31 provideAdapter

It is recommend to use registerAdapter .

8.32 provideHandler

It is recommend to use registerHandler .

8.33 provideSubscriptionAdapter

It is recommend to use registerSubscriptionAdapter .

8.34 provideUtility

It is recommend to use registerUtility .

8.35 providedBy

Test whether the interface is implemented by the object. Return true if the object asserts that it implements the interface, including asserting that it implements an extended interface.

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

Example 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
```

Example 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"

To get a list of all interfaces provided by that object::

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

8.36 queryAdapter

Look for a named adapter to an interface for an object. Returns an adapter that can adapt object to interface. If a matching adapter cannot be found, returns the default.

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

Example:

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

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

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True
```

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

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

8.37 queryAdapterInContext

Instead of this function, use *context* argument of queryAdapter function.

- Location: `zope.component`
- Signature: *queryAdapterInContext(object, interface, context, default=None)*

Example:

```
>>> 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 IRegistrar(Interface):
...     """A registrar will register object's details"""
...
...     def register():
...         """Register object's details"""
...

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

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

```
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> sm.registerAdapter(GuestRegistrarNG,
...                    (IGuest,), IRegistrar)

>>> from zope.component import getAdapterInContext
>>> from zope.component import queryAdapterInContext

>>> queryAdapterInContext(jack, IRegistrar, sm) #doctest: +ELLIPSIS
<GuestRegistrarNG object at ...>
```

8.38 queryMultiAdapter

Look for a multi-adapter to an interface for objects. Returns a multi-adapter that can adapt objects to interface. If a matching adapter cannot be found, returns the default. The name consisting of an empty string is reserved for unnamed adapters. The unnamed adapter methods will often call the named adapter methods with an empty string for a name.

- Location: `zope.component`
- Signature: *queryMultiAdapter(objects, interface=Interface, name=u'', default=None, context=None)*

Example:

```
>>> 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 ...>
```

8.39 queryUtility

Look up a utility that provides an interface. If one is not found, returns default.

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

Example:

```
>>> 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'
```

8.40 registerAdapter

Register an adapter factory.

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

Example:

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

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

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
```

```
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

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

You can test it like this:

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

8.41 registeredAdapters

Return an iterable of *IAdapterRegistrations*. These registrations describe the current adapter registrations in the object.

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

Example:

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

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

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

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

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


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

8.42 registeredHandlers

Return an iterable of *IHandlerRegistrations*. These registrations describe the current handler registrations in the object.

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

Example:

```
>>> 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
```

8.43 registeredSubscriptionAdapters

Return an iterable of *ISubscriptionAdapterRegistrations*. These registrations describe the current subscription adapter registrations in the object.

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

Example:

```
>>> 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
```

8.44 registeredUtilities

Return an iterable of *IUtilityRegistrations* . These registrations describe the current utility registrations in the object.

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

Example:

```
>>> 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
```

8.45 registerHandler

Register a handler. A handler is a subscriber that doesn't compute an adapter but performs some function when called.

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

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

Example:

```
>>> 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'
```

8.46 registerSubscriptionAdapter

Register a subscriber factory.

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

Example:

```
>>> 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)
```

8.47 registerUtility

Register a utility.

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

Example:

```
>>> 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)
```

8.48 subscribers

Get subscribers. Subscribers are returned that provide the provided interface and that depend on and are computed from the sequence of required objects.

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

Example:

```
>>> 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']
```

8.49 unregisterAdapter

Register an adapter factory. A boolean is returned indicating whether the registry was changed. If the given component is None and there is no component registered, or if the given component is not None and is not registered, then the function returns False, otherwise it returns True.

- Location: `zope.component - IComponentRegistry`
- Signature: *unregisterAdapter(factory=None, required=None, provided=None, name=u'')*

Example:

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

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

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

>>> class GuestRegistrarNG(object):
...
...     implements(IRegistrar)
...     adapts(IGuest)
...
...     def __init__(self, guest):
...         self.guest = guest
...
...     def register(self):
...         next_id = get_next_id()
...         guests_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_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(GuestRegistrarNG,
...                     (IGuest,), IRegistrar, 'ng6')

You can test it like this:

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

Now unregister:

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

After unregistration:

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

8.50 unregisterHandler

Unregister a handler. A handler is a subscriber that doesn't compute an adapter but performs some function when called. A boolean is returned indicating whether the registry was changed.

- Location: `zope.component - IComponentRegistry`
- Signature: *unregisterHandler(handler=None, required=None, name=u'')*

Example:

```
>>> 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
```

8.51 unregisterSubscriptionAdapter

Unregister a subscriber factory. A boolean is returned indicating whether the registry was changed. If the given component is None and there is no component registered, or if the given component is not None and is not registered, then the function returns False, otherwise it returns True.

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

Example:

```
>>> 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']
```

Now unregister:

```
>>> gsm.unregisterSubscriptionAdapter(AdequateLength)
True

After unregistration:

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

8.52 unregisterUtility

Unregister a utility. A boolean is returned indicating whether the registry was changed. If the given component is None and there is no component registered, or if the given component is not None and is not registered, then the function returns False, otherwise it returns True.

- Location: `zope.component - IComponentRegistry`
- Signature: *unregisterUtility(component=None, provided=None, name=u'')*

Example:

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

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

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

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

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

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

Now unregister:

>>> gsm.unregisterUtility(greet)
True

After unregistration:
```

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