A Comprehensive Guide to Zope Component Architecture

Baiju M

Version: 0.5.7
Printed Book: http://www.lulu.com/content/1561045
Online PDF: http://www.muthukadan.net/docs/zca.pdf

Copyright (C) 2007,2008,2009 Baiju M <baiju.m.mail AT gmail.com>.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or (at your option) any later version published by the Free Software Foundation.

The source code in this document is subject to the provisions of the Zope Public License, Version 2.1 (ZPL).

THE SOURCE CODE IN THIS DOCUMENT AND THE DOCUMENT ITSELF IS PROVIDED "AS IS" AND ANY AND ALL EXPRESS OR IMPLIED WARRANTIES ARE DISCLAIMED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF TITLE, MERCHANTABILITY, AGAINST INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgements

Many people have helped me to write this book. The initial draft was reviewed by my colleague Brad Allen. When I announced this book through my blog, I received many encouraging comments to proceed with this work. Kent Tenney edited most parts of the book, he also rewrote the example application. Many others sent me fixes and comments including, Lorenzo Gil Sanchez, Michael Haubenwallner, Nando Quintana, Stephane Klein, Tim Cook, Kamal Gill and Thomas Herve. Lorenzo translated this work to Spanish and Stephane translated it to French. Thanks to all !

Contents

1 Getting started **5**
1.1 Introduction 5
1.2 A brief history 6
1.3 Installation 6
1.4 Experimenting with code 7

2 An example **9**
2.1 Introduction 9
2.2 Procedural approach 9
2.3 Object oriented approach 10
2.4 The adapter pattern 12

3 Interfaces **13**
3.1 Introduction 13
3.2 Declaring interfaces 14
3.3 Implementing interfaces 15
3.4 Example revisited 16
3.5 Marker interfaces 16
3.6 Invariants 17

4 Adapters **19**
4.1 Implementation 19
4.2 Registration 20
4.3 Querying adapter 21
4.4 Retrieving adapter using interface 22
4.5 Adapter pattern 22

5 Utility **25**
5.1 Introduction 25
5.2 Simple utility 25
5.3 Named utility 26
5.4 Factory 27

6 Advanced adapters 29
6.1 Multi adapter . . . 29
6.2 Subscription adapter . . . 30
6.3 Handler . . . 32

7 ZCA usage in Zope 35
7.1 ZCML . . . 35
7.2 Overrides . . . 36
7.3 NameChooser . . . 38
7.4 LocationPhysicallyLocatable . . . 38
7.5 DefaultSized . . . 39
7.6 ZopeVersionUtility . . . 39

8 Case study 41
8.1 Introduction . . . 41
8.2 Use cases . . . 41
8.3 Overview of PyGTK code . . . 43
8.4 The code . . . 44
8.5 PySQLite . . . 53
8.6 ZODB . . . 53
8.7 Conclusions . . . 53

9 Reference 55
9.1 adaptedBy . . . 55
9.2 adapter . . . 55
9.3 adapts . . . 56
9.4 alsoProvides . . . 57
9.5 Attribute . . . 58
9.6 classImplements . . . 58
9.7 classImplementsOnly . . . 59
9.8 classProvides . . . 60
9.9 ComponentLookupError . . . 60
9.10 createObject . . . 60
9.11 Declaration . . . 61
9.12 directlyProvidedBy . . . 61
9.13 directlyProvides . . . 62
9.14 getAdapter . . . 64
9.15 getAdapterInContext . . . 65
9.16 getAdapters . . . 66

9.17 getAllUtilitiesRegisteredFor . . . 67
9.18 getFactoriesFor . . . 68
9.19 getFactoryInterfaces . . . 69
9.20 getGlobalSiteManager . . . 69
9.21 getMultiAdapter . . . 70
9.22 getSiteManager . . . 71
9.23 getUtilitiesFor . . . 72
9.24 getUtility . . . 72
9.25 handle . . . 73
9.26 implementedBy . . . 74
9.27 implementer . . . 75
9.28 implements . . . 75
9.29 implementsOnly . . . 76
9.30 Interface . . . 77
9.31 moduleProvides . . . 77
9.32 noLongerProvides . . . 78
9.33 provideAdapter . . . 79
9.34 provideHandler . . . 79
9.35 provideSubscriptionAdapter . . . 79
9.36 provideUtility . . . 79
9.37 providedBy . . . 79
9.38 queryAdapter . . . 80
9.39 queryAdapterInContext . . . 82
9.40 queryMultiAdapter . . . 83
9.41 queryUtility . . . 84
9.42 registerAdapter . . . 85
9.43 registeredAdapters . . . 86
9.44 registeredHandlers . . . 88
9.45 registeredSubscriptionAdapters . . . 89
9.46 registeredUtilities . . . 90
9.47 registerHandler . . . 90
9.48 registerSubscriptionAdapter . . . 91
9.49 registerUtility . . . 92
9.50 subscribers . . . 93
9.51 unregisterAdapter . . . 95
9.52 unregisterHandler . . . 96
9.53 unregisterSubscriptionAdapter . . . 97
9.54 unregisterUtility . . . 99

Chapter 1

Getting started

1.1 Introduction

Developing a large software system is always very complicated. An object oriented approach to analysis, design and programming has been shown to be well suited for dealing with large systems. Component based design, and programming using components are becoming very popular these days. Component based approach helps you to write and maintain easily unit-testable software systems. There are many frameworks for supporting component based design in different languages, some are even language neutral. Examples of these are Microsoft's COM and Mozilla's XPCOM.

Zope Component Architecture (ZCA) is a Python framework for supporting component based design and programming. It is very well suited to developing large Python software systems. The ZCA is not specific to the Zope web application server: it can be used for developing any Python application. Maybe it should be called as *Python Component Architecture*.

The ZCA is all about using Python objects effectively. Components are reusable objects with introspectable interfaces. An interface is an object that describes how you work with a particular component. In other words, component provides an interface implemented in a class, or any other callable object. It doesn't matter how the component is implemented, the important part is that it comply with its interface contracts. Using ZCA, you can spread the complexity of systems over multiple cooperating components. It helps you to create two basic kinds of components: *adapter* and *utility*.

There are three core packages related to the ZCA:

- `zope.interface` is used to define the interface of a component.
- `zope.event` provides a simple event system.
- `zope.component` deals with creation, registration and retrieval of components.

Remember, the ZCA is not the components themselves, rather it is about creating, registering, and retrieving components. Remember also, an *adapter* is a normal Python class (or a factory in general) and *utility* is a normal Python callable object.

The ZCA framework is developed as part of the Zope 3 project. As noted earlier, it is a pure Python framework, so it can be used in any kind of Python application. Currently Zope 3, Zope 2 and Grok projects use this framework extensively. There are many other projects including non-web applications using it[1].

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

1.2 A brief history

The ZCA framework project began in 2001 as part of Zope 3 project. It grew out of lessons learned while developing large software systems using Zope 2. Jim Fulton was the project leader of this project. Many people contributed to the design and implementation, including but not limited to, Stephan Richter, Philipp von Weitershausen, Guido van Rossum (*aka. Python BDFL*), Tres Seaver, Phillip J Eby and Martijn Faassen.

Initially ZCA defined additional components; *services* and *views*, but the developers came to realize that utility can replace *service* and multi-adapter can replace *view*. Now ZCA has a very small number of core component types: *utilities*, *adapters*, *subscribers* and *handlers*. In fact, *subscribers* and *handlers* are two special types of adapters.

During the Zope 3.2 release cycle, Jim Fulton proposed a major simplification of ZCA[2]. With this simplification, a new single interface (*IComponentRegistry*) for registration of both global and local component was created.

The `zope.component` package had a long list of dependencies, many of which were not required for a non Zope 3 application. During PyCon 2007, Jim Fulton added setuptools' *extras_require* feature to allow separating out core ZCA functionality from add-on features[3].

In March 2009, Tres Seaver removed dependcencies of `zope.deferredimport` and `zope.proxy`.

Now, The ZCA project is an independent project with it's own release cycle and Subversion repository. This project is coming as part the bigger the Zope framework project[4]. However, issues and bugs are still tracked as part of the Zope 3 project[5], and the main zope-dev list is used for development discussions[6]. There is also another general user list for Zope 3 (*zope3-users*) which can be used for any queries about the ZCA[7].

1.3 Installation

The `zope.component`, package together with the `zope.interface` and `zope.event` packages are the core of Zope component architecture. They provide facilities for defining, registering and looking up components. The `zope.component` package and its dependencies are available in egg format from the Python Package Index (PyPI)[8].

You can install `zope.component` and it's dependencies using *easy_install*[9]

```
$ easy_install zope.component
```

This command will download `zope.component` and its dependencies from PyPI and install it in your Python path.

Alternately, you can download `zope.component` and its dependencies from PyPI and then install them. Install packages in the order given below. On Windows, you may need binary packages of `zope.interface`.

1. `zope.interface`
2. `zope.event`
3. `zope.component`

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

To install these packages, after downloading them, you can use `easy_install` command with argument as the eggs. (You may also give all these eggs in the same line.):

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

You can also install these packages after extracting each one separately. For example:

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

These methods will install the ZCA to the *system Python*, in the `site-packages` directory, which can cause problems. In a Zope 3 mailing list post, Jim Fulton recommends against using the system Python[10]. You can use `virtualenv` and/or `zc.buildout` for playing with any Python packages, also good for deployments.

1.4 Experimenting with code

There are two approaches in Python for setting up isolated working environments for developing Python applications. `virtualenv` created by Ian Biking and `zc.buildout` created by Jim Fulton are these two packages. You can also use these packages together. Using these packages you can install `zope.component` and other dependencies into an isolated working environment. This is a good practice for experimenting with any Python code, and familiarity with these tools will be beneficial when developing and deploying applications.

virtualenv

You can install `virtualenv` using `easy_install`:

```
$ easy_install virtualenv
```

Then create a new environment like this:

```
$ virtualenv --no-site-packges myve
```

This will create a new virtual environment in the `myve` directory. Now, from inside the `myve` directory, you can install `zope.component` and dependencies using `easy_install` inside `myve/bin` directory:

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

Now you can import `zope.interface` and `zope.component` from the new `python` interpreter inside `myve/bin` directory:

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

This command will give you a Python prompt which you can use to run the code in this book.

zc.buildout

Using `zc.buildout` with `zc.recipe.egg` recipe you can create Python interpreter with specified Python eggs. First, install `zc.buildout` using `easy_install` command. (You may also do it inside virtual environment). To create new buildout to experiment with Python eggs, first create a directory and initialize it using `buildout init` command:

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

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

Now the new `mybuildout` directory is a buildout. The default configuration file for buildout is *buildout.cfg* . After initializing, it will be having this content:

```
[buildout]
parts =
```

You can change it like this:

```
[buildout]
parts = py

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

Now run `buildout` command available inside `mybuildout/bin` directory without any argument. This will create a new Python interpreter inside `mybuildout/bin` directory:

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

This command will give you a Python prompt which you can use to run the code in this book.

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. We will generate unique id's for the dictionary, the associated value will be a dictionary of details about the booking.

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

A minimal implementation requires a function which we pass the details of the booking, and a supporting function which provides the the unique id for the storage dictionary key.

We can get the unique id like this:

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

As you can see, the *get_next_id* function implementation is very simple. The function gets a list of keys and checks for an empty list. If the list is empty this is our first booking, so we return *1*. If the list is not empty, we add *1* to the maximum value in the list and return it.

Now we will use the above function to create entries in the bookings_db dictionary:

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

The requirements of a hotel booking management application require considering additional data:

- phone numbers
- room options
- payment methods
- ...

And code to manage the data:

- cancel a reservation
- update a reservation
- pay for a room
- persist the data
- insure security of the data
- ...

Were we to continue with the procedural example, we would create many functions, passing data back and forth between them. As requirements change and are added, the code becomes harder to maintain and bugs become harder to find and fix.

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

2.3 Object oriented approach

Our discussion of object oriented design will introduce the *class* which serves to encapsulate the data, and the code to manage it.

Our main class will be the *FrontDesk*. FrontDesk, or other classes it delegates to, will know how to manage the data for the hotel. We will create *instances* of FrontDesk to apply this knowledge to the business of running a hotel.

Experience has shown that by consolidating the code and data requirements via objects, we will end up with a design which is easier to understand, test, and change.

Lets look at the implementation details of a FrontDesk class:

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

In this implementation, the *frontdesk* object (an instance of *FrontDesk* class) is able to handle the bookings. We can use it like this:

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

Any real project will involve changing requirements. In this case management has decided that each guest must provide a phone number, so we must change the code.

We can achieve this requirement by adding one argument to the *book_room* method which will be added to the dictionary of values:

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

In addition to migrating the data to new schema, we now have to change all calls to FrontDesk. If we abstract the details of guest into an object and use it for registration, the code changes can be minimized. We now can make changes to the details of the guest object and the calls to FrontDesk won't need to change.

Now we have:

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

We still will have to change code to respond to changing requirements. This is unavoidable, however, our goal is to minimize those changes, thereby increasing maintainability.

> **Note**
>
> When coding, it is important to feel free to make changes without fear of breaking the application. The way to get the immediate feedback required is via automated testing. With well written tests (and good version control) you can make changes large or small with impunity. A good source of information about this programming philosophy is the book *Extreme Programming Explained* by Kent Beck.

By introducing the guest object, you saved some typing. More importantly, the abstraction provided by the guest object made the system simpler and more understandable. As a result, the code is easier to restructure and maintain.

2.4 The adapter pattern

In a real application, the frontdesk object will need to handle chores such as cancellations and updates. In the current design, we will need to pass the guest object to frontdesk every time we call methods such as *cancel_booking* and *update_booking*.

We can avoid this requirement if we pass the guest object to FrontDesk.__init__(), making it an attribute of the instance.

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

The solution we have reached is a well known pattern, the *adapter*. In general, an adapter *contains* an *adaptee*:

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

This pattern will be useful in dealing with implementation details which depend on considerations such as:

- changing customer requirements
- storage requirements (ZODB, RDBM, XML ...)
- output requirements (HTML, PDF, plain text ...)
- markup rendering (ReST, Markdown, Textile ...)

ZCA uses adapters and a *component registry* to provide the capability to change implementation details of code via *configuration*.

As we will see in the section on ZCA adapters, the ability to configure implementation details provides useful capability:

- the ability to switch between implementations
- the ability to add implementations as needed
- increased re-use of both legacy and ZCA code

These capabilities lead to code that is flexible, scalable and re-usable. There is a cost however, maintaining the component registry adds a level of complexity to the application. If an application will never require these features, ZCA is unnecessary.

We are now ready to begin our study of the Zope Component Architecture, beginning with interfaces.

Chapter 3

Interfaces

3.1 Introduction

The README.txt[11] in path/to/zope/interface defines interfaces like this:

```
Interfaces are objects that specify (document) the external behavior
of objects that "provide" them.  An interface specifies behavior
through:

- Informal documentation in a doc string

- Attribute definitions

- Invariants, which are conditions that must hold for objects that
  provide the interface
```

The classic software engineering book *Design Patterns*[12] by the *Gang of Four* recommends that you "Program to an interface, not an implementation". Defining a formal interface is helpful in understanding a system. Moreover, interfaces bring to you all the benefits of ZCA.

An interface specifies the characteristics of an object, it's behaviour, it's capabilities. The interface describes *what* an object can do, to learn *how*, you must look at the implementation.

Commonly used metaphors for interfaces are *contract* or *blueprint*, the legal and architectural terms for a set of specifications.

In some modern programming languages: Java, C#, VB.NET etc, interfaces are an explicit aspect of the language. Since Python lacks interfaces, ZCA implements them as a meta-class to inherit from.

Here is a classic *hello world* style example:

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

[11] The Zope code tree is full of README.txt files which offer wonderful documentation.

[12] http://en.wikipedia.org/wiki/Design_Patterns

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 your code uses. If you want to examine implementation details you need to access the class `Host`, either via the source code or an API[13] documentation tool.

Now we will begin to use the ZCA interfaces. 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 inherits from zope.interface.Interface. This use (abuse?) of Python's class statement is how ZCA defines an interface. The `I` prefix for the interface name is a useful 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.

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

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 frontdesk object:

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

>>> class IDesk(Interface):
...     """A frontdesk 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 frontdesk interface defined here is `IDesk`. 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.

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 *FrontDeskNG* adapter with explicit interface declaration:

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

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

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

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

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

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

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

The *FrontDeskNG* 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(FrontDeskNG,
...                     (IGuest,), IDesk, '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 implemented 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(FrontDeskNG, 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 *FrontDeskNG* 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 *IDesk* 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 *IDesk* interface also with name as 'ng'. Here both `getAdapter` and `queryAdapter` works similarly:

```
>>> getAdapter(jack, IDesk, 'ng') #doctest: +ELLIPSIS
<FrontDeskNG object at ...>
>>> queryAdapter(jack, IDesk, 'ng') #doctest: +ELLIPSIS
<FrontDeskNG 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, IDesk, 'not-exists') #doctest: +ELLIPSIS
Traceback (most recent call last):
...
ComponentLookupError: ...
>>> reg = queryAdapter(jack,
...            IDesk, '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, IDesk) #doctest: +ELLIPSIS
Traceback (most recent call last):
...
ComponentLookupError: ...
>>> reg = queryAdapter(jack, IDesk) #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.

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

Keyword name can be omitted:

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

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

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

Here *FrontDeskNG* is registered without name:

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

Now the adapter lookup should succeed:

```
>>> IDesk(jack, 'default-output') #doctest: +ELLIPSIS
<FrontDeskNG 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 Design Patterns book, GoF says: "Often the adapter is responsible for functionality the adapted class doesn't provide". ZCA adapter has more focus on adding functionality 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.)[14]

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:

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

"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.

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 the ZCA 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

A utility can be registered with a name or without a name. A utility registered with a name is called named utility, which you will see in the next section. 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"""
```

Like an adapter a utility may have more than one implementation. Here is a possible implementation of the above interface:

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

The actual utility will be an instance of this class. To use this utility, you have to register it, later you can query it using the ZCA API. You can register an instance of this class (*utility*) 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. Only once you are creating the instance of a utility class, but adapter instances are dynamically created whenever you query for it.

5.3 Named utility

When registering a utility component, like adapter, you can use a name. As mentioned in the previous section, a utility registered with a particular name is called named utility.

This is how you can register the *greeter* utility with a name:

```
>>> 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 up the interface 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.

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. The adapter look up functions, *getAdapter* and *queryAdapter* also works similarly.

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

This chapter discuss some advanced adapters like multi adapter, subscription adapter and handler.

6.1 Multi adapter

A simple adapter normally adapts only one object, but an adapter may adapt more than one object. If an adapter adapts more than one objects, it is called *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 interface. 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, 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 will go through usage of the ZCA 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 utility, 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

Case study

> **Note**
>
> This chapter is not yet completed. Please send your suggestions !

8.1 Introduction

This chapter demonstrates creating a desktop application using PyGTK GUI library and the ZCA. This application also use two different kinds of data persistence mechanisms, one object database (ZODB) & and another relational database (SQLite). However, practically, only one storage can be used for a particular installation. The reason for using two different persistence mechanisms is to demonstrate how to use the ZCA to glue components. Majority of the code in this application is related to PyGTK.

As the application grows you may use the ZCA components wherever you want pluggability or extensibility. Use plain Python objects directly where you do not required pluggability or extensibility.

There is no difference in using ZCA for web or desktop or any other kind of application or framework. It is better to follow a convention for the location from where you are going to register components. This application use a convention, which can be extended by putting registration of similar components in separate modules and later import them from main registration module. In this application the main component registration module is *register.py*.

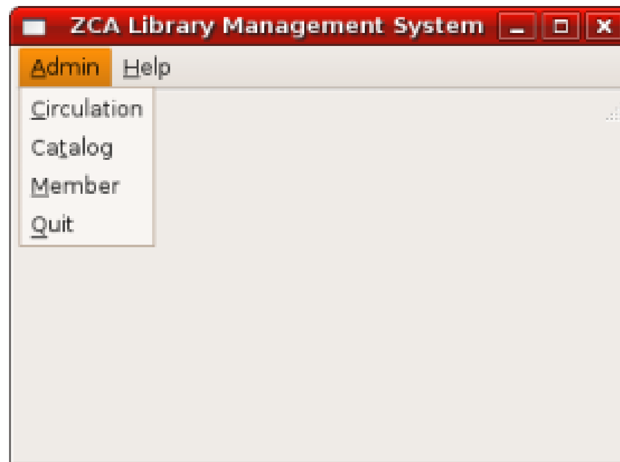
Source code of this application can be downloaded from: http://www.muthukadan.net/downloads/zcalib.tar.bz2

8.2 Use cases

The application we are going to discuss here is a library management system with minimal features. The requirements can be summarized like this:

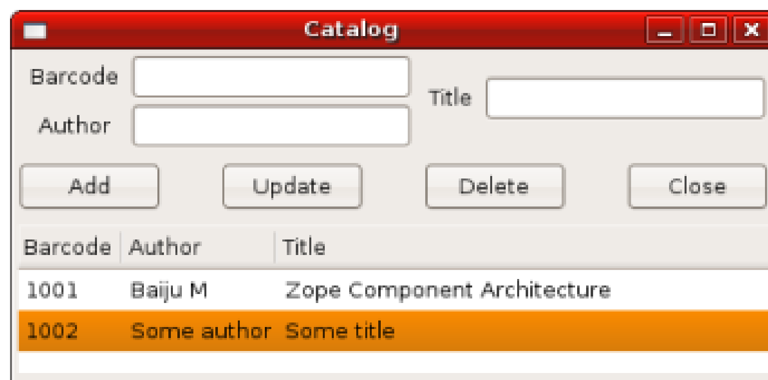
- Add members with a unique number and name.
- Add books with barcode, author & title
- Issue books
- Return books

The application can be designed in such a way that major features can be accessed from a single window. The main window for accessing all these features can be designed like this:

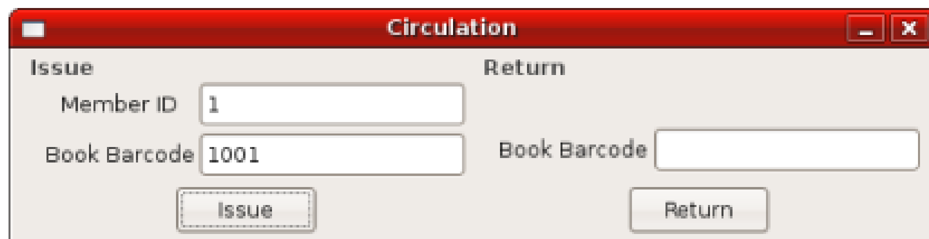


From member window, user should be able to manage members. So, it should be possible to *add*, *update* and *delete* members as shown in the below picture:

Similar to member window, the catalog window allows user to *add*, *edit* and *delete* books:



The circulation window should have the facility for issuing and returning books:



8.3 Overview of PyGTK code

As you can see in the code, most of the code are related to PyGTK. The code structure is very similar for different windows. The windows of this application are designed using Glade GUI builder. You should give proper names for widgets you are going to use from code. In the main window, all menu entries has names like: circulation, catalog, member, quit & about.

The `gtk.glade.XML` class is used to parse glade file, this will create GUI widget objects. This is how to parse and access objects:

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

In the mainwindow.py, you can see code like this:

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

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

The name of main window widget is *mainwindow*. Similarly, other widgets are retrieved like this:

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

Then, these widgets are connected for some events:

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

The *delete_event* is the event when the window is closing using the window close button. The *activate* event is emitted when the menu is selected. The widgets are connected to some callback functions for some events.

You can see from the above code that, main window is connected to *on_delete_event* method for *delete_event*. The *quit* widget is also connected to same method for *activate* event:

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

The callback function just call *main_quit* function

8.4 The code

This is the *zcalib.py*:

```
import registry
import mainwindow

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

Here, two modules are imported *registry* and *mainwindow*. Then, registry is initialized and mainwindow's *main* function is called. If user is trying to exit application using *Ctrl+C*, system will exit normally, that's why we catched *KeyboardInterrupt* exception.

This is the *registry.py*:

```
import sys
from zope.component import getGlobalSiteManager

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


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

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

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

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

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

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

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

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

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

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

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

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

Look at the *initialize* function which we are calling from the main module, *zcalib.py*. The *initialize* function first check which db to use, relational database (RDB) or object database (ODB) and this checking is done at *check_use_relational_db* function. If *-r* option is given at command line, it will call *initialize_rdb* otherwise, *initialize_odb*. If the RDB function is called, it will setup all components related to RDB. On the other hand, if the ODB function is called, it will setup all components related to ODB.

Here is the *mainwindow.py*:

```
import os
import gtk
import gtk.glade

from circulationwindow import circulationwindow
```

```
from catalogwindow import catalogwindow
from memberwindow import memberwindow

class MainWindow(object):

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

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

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

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

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

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

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

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

    def on_about_activate(self, *args):
        pass

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

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

The *main* function here creates an instance of *MainWindow* class, which will initialize all widgets.

Here is the *memberwindow.py*:

```
import os
```

```
import gtk
import gtk.glade

from zope.component import getAdapter

from components import Member
from interfaces import IDbOperation


class MemberWindow(object):

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

        self.memberwindow = xmlobj.get_widget('memberwindow')
        self.number = xmlobj.get_widget('number')
        self.name = xmlobj.get_widget('name')
        add = xmlobj.get_widget('add')
        update = xmlobj.get_widget('update')
        delete = xmlobj.get_widget('delete')
        close = xmlobj.get_widget('close')
        self.treeview = xmlobj.get_widget('treeview')

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

        self.initialize_list()

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

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

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

    def initialize_list(self):
        self.list_store = gtk.ListStore(object, str, str)
```

```
        self.treeview.set_model(self.list_store)
        tvcolumn = gtk.TreeViewColumn('Member Number')
        self.treeview.append_column(tvcolumn)

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

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

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

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

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

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

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

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

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

memberwindow = MemberWindow()
```

Here is the *components.py*:

```
from zope.interface import implements

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

class Book(object):

    implements(IBook)

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

class Member(object):

    implements(IMember)

    number = ""
    name = ""

class Circulation(object):

    implements(ICirculation)

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

Here is the *interfaces.py*:

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


class IBook(Interface):

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


class IMember(Interface):

    number = Attribute("ID number")
```

```
    name = Attribute("Name of member")


class ICirculation(Interface):

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


class IRelationalDatabase(Interface):

    def commit():
        pass

    def rollback():
        pass

    def cursor():
        pass

    def get_next_id():
        pass


class IObjectDatabase(Interface):

    def commit():
        pass

    def rollback():
        pass

    def container():
        pass

    def get_next_id():
        pass


class IDbOperation(Interface):

    def get():
        pass

    def add():
        pass

    def update():
        pass

    def delete():
        pass
```

Here is the *member.py*:

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

from components import Member

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


class MemberRDbOperation(object):

    implements(IDbOperation)
    adapts(IMember)

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

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

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

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

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


class MemberODbOperation(object):

    implements(IDbOperation)
    adapts(IMember)

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

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

```
            return members.values()

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

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

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

8.5 PySQLite

8.6 ZODB

8.7 Conclusions

Chapter 9

Reference

9.1 adaptedBy

This function helps to find the adapted interfaces.

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

Example:

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

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

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

9.2 adapter

Adapters can be any callable object, you can use the *adapter* decorator to declare that a callable object adapts some interfaces (or classes)

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

Example:

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

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

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

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

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

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

>>> jack = Person()
>>> jack.name = "Jack"
>>> jack.job = Job()
>>> personJob(jack) #doctest: +ELLIPSIS
<Job object at ...>
```

9.3 adapts

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

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

9.4 alsoProvides

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(IDesk)
...     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
```

9.5 Attribute

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

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

Example:

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

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

9.6 classImplements

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(IDesk)
...     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
```

9.7 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
```

9.8 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
```

9.9 ComponentLookupError

This is the exception raised when a component lookup fails.

Example:

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

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

9.10 createObject

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 ...>
```

9.11 Declaration

Need not to use directly.

9.12 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
```

9.13 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
```

9.14 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 IDesk(Interface):
...     """A frontdesk will register object’s details"""
...
...     def register():
...         """Register object’s details"""
...

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

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

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

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

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

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

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

9.15 getAdapterInContext

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

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

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

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

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

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

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

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

>>> IDesk.providedBy(jack_frontdesk)
True

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

>>> from zope.component import getAdapterInContext

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

9.16 getAdapters

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

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

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

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

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

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

9.17 getAllUtilitiesRegisteredFor

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 ...>]
```

9.18 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'>>)]
```

9.19 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>
```

9.20 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
```

9.21 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)*
- See also: queryMultiAdapter

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 ...>
```

9.22 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
```

9.23 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 ...>)]
```

9.24 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'
```

9.25 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'
```

9.26 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']
```

9.27 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>]
```

9.28 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
```

9.29 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
```

9.30 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"""
```

9.31 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)*
- See also: directlyProvides

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

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

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

9.32 noLongerProvides

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
```

9.33 provideAdapter

It is recommended to use registerAdapter .

9.34 provideHandler

It is recommended to use registerHandler .

9.35 provideSubscriptionAdapter

It is recommended to use registerSubscriptionAdapter .

9.36 provideUtility

It is recommended to use registerUtility .

9.37 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']
```

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

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

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

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

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

>>> IDesk.providedBy(jack_frontdesk)
True

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

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

9.39 queryAdapterInContext

Look for a special adapter to an interface for an object.

NOTE: This method should only be used if a custom context needs to be provided to provide custom component lookup. Otherwise, call the interface, as in:

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

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

Context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

If the object has a __conform__ method, this method will be called with the requested interface. If the method returns a non-None value, that value will be returned. Otherwise, if the object already implements the interface, the object will be returned.

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

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

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

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

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

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

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

>>> IDesk.providedBy(jack_frontdesk)
True

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

>>> from zope.component import queryAdapterInContext

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

9.40 queryMultiAdapter

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)*
- See also: getMultiAdapter

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 ...>
```

9.41 queryUtility

This function is used to 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'
```

9.42 registerAdapter

This function is used to register an adapter factory.

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

Example:

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

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

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

>>> class FrontDeskNG(object):
```

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

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

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

>>> IDesk.providedBy(jack_frontdesk)
True

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

You can test it like this:

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

9.43 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 IDesk(Interface):
...     """A frontdesk will register object’s details"""
...
...     def register():
...         """Register object’s details"""
...

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

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

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

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

>>> IDesk.providedBy(jack_frontdesk)
True

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


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

9.44 registeredHandlers

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
```

9.45 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
```

9.46 registeredUtilities

This function 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
```

9.47 registerHandler

This function is used to 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='')*
- See also: unregisterHandler

Note: 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'
```

9.48 registerSubscriptionAdapter

This function is used to register a subscriber factory.

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

- See also: unregisterSubscriptionAdapter

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)
```

9.49 registerUtility

This function is used to register a utility.

- Location: `zope.component - IComponentRegistry`

- Signature: *registerUtility(component, provided=None, name=u”, info=u”)*
- See also: unregisterUtility

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)
```

9.50 subscribers

This function is used to 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']
```

9.51 unregisterAdapter

This function is used to unregister 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")*
- See also: registerAdapter

Example:

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

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

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

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

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

```
...     implements(IGuest)
...
...     def __init__(self, name, place):
...         self.name = name
...         self.place = place

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

>>> IDesk.providedBy(jack_frontdesk)
True

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

You can test it like this:

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

Now unregister:

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

After unregistration:

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

9.52 unregisterHandler

This function is used for unregistering 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")*
- See also: registerHandler

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
```

9.53 unregisterSubscriptionAdapter

This function is used to 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'')*
- See also: registerSubscriptionAdapter

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()]
[]
```

9.54 unregisterUtility

This function is used for unregistering 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")*
- See also: registerUtility

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
```