# Arquitectura de Componentes de Zope

**Autor**: Baiju M
**Version**: 0.4.2
**URL original**: http://www.muthukadan.net/docs/zca.pdf
**Traductor**: Lorenzo Gil Sanchez <lgs@sicem.biz>
**URL en español**: http://www.muthukadan.net/docs/zca-es.pdf



**Note**

Gracias a Kent Tenney (Wisconsin, USA) y Brad Allen (Dallas, USA) por sus sugerencias.

# Contents


**1 Primeros pasos** **7**

1.1 Introducción . . . 7
1.2 Una breve historia . . . 7
1.3 Instalación . . . 8
1.4 Experimentando con código . . . 9

**2 Un ejemplo** **11**

2.1 Introducción . . . 11
2.2 Enfoque procedural . . . 11
2.3 Enfoque orientado a objetos . . . 12
2.4 El patrón adaptador . . . 13

**3 Interfaces** **15**

3.1 Introducción . . . 15
3.2 Declarando interfaces . . . 16
3.3 Implementando interfaces . . . 17
3.4 Ejemplo revisado . . . 17
3.5 Interfaces de marcado . . . 18
3.6 Invariantes . . . 18

**4 Adaptadores** **21**

4.1 Implementación . . . 21
4.2 Registro . . . 22
4.3 Patrón de consulta . . . 23
4.4 Recuperar un adaptador usando una interfaz . . . 24
4.5 Patrón de Adaptador . . . 25

**5 Utilidad** **27**

5.1 Introducción . . . 27
5.2 Utilidad simple . . . 27
5.3 Utilidad con nombre . . . 28
5.4 Fábrica . . . 29

**6 Adaptadores avanzados** **31**
6.1 Multi adaptador . . . 31
6.2 Adaptador de subscripción . . . 32
6.3 Manejador . . . 34

**7 Uso de la ZCA en Zope** **37**
7.1 ZCML . . . 37
7.2 Redefiniciones . . . 38
7.3 NameChooser (Elejidor de nombres) . . . 40
7.4 LocationPhysicallyLocatable . . . 40
7.5 DefaultSized . . . 41
7.6 ZopeVersionUtility . . . 41

**8 Referencia** **43**
8.1 Attribute . . . 43
8.2 Declaration . . . 43
8.3 Interface . . . 43
8.4 adapts . . . 44
8.5 alsoProvides . . . 44
8.6 classImplements . . . 45
8.7 classImplementsOnly . . . 46
8.8 classProvides . . . 47
8.9 ComponentLookupError . . . 47
8.10 createObject . . . 47
8.11 directlyProvidedBy . . . 48
8.12 directlyProvides . . . 49
8.13 getAdapter . . . 50
8.14 getAdapterInContext . . . 52
8.15 getAdapters . . . 53
8.16 getAllUtilitiesRegisteredFor . . . 54
8.17 getFactoriesFor . . . 55
8.18 getFactoryInterfaces . . . 55
8.19 getGlobalSiteManager . . . 56
8.20 getMultiAdapter . . . 56
8.21 getSiteManager . . . 58
8.22 getUtilitiesFor . . . 58
8.23 getUtility . . . 59
8.24 handle . . . 60

8.25 implementedBy . . . . . 61
8.26 implementer . . . . . 62
8.27 implements . . . . . 62
8.28 implementsOnly . . . . . 63
8.29 moduleProvides . . . . . 64
8.30 noLongerProvides . . . . . 64
8.31 provideAdapter . . . . . 65
8.32 provideHandler . . . . . 65
8.33 provideSubscriptionAdapter . . . . . 65
8.34 provideUtility . . . . . 65
8.35 providedBy . . . . . 65
8.36 queryAdapter . . . . . 67
8.37 queryAdapterInContext . . . . . 68
8.38 queryMultiAdapter . . . . . 69
8.39 queryUtility . . . . . 70
8.40 registerAdapter . . . . . 71
8.41 registeredAdapters . . . . . 72
8.42 registeredHandlers . . . . . 74
8.43 registeredSubscriptionAdapters . . . . . 75
8.44 registeredUtilities . . . . . 76
8.45 registerHandler . . . . . 76
8.46 registerSubscriptionAdapter . . . . . 77
8.47 registerUtility . . . . . 78
8.48 subscribers . . . . . 79
8.49 unregisterAdapter . . . . . 81
8.50 unregisterHandler . . . . . 82
8.51 unregisterSubscriptionAdapter . . . . . 83
8.52 unregisterUtility . . . . . 85

# Chapter 1

# Primeros pasos

## 1.1 Introducción

Desarrollar un sistema software grande es siempre muy complicado. Se ha visto que un enfoque orientado a objetos para el análisis, diseño y programación funciona bien al tratar con sistemas grandes. El diseño basado en componentes, y la programación utilizando componentes se están haciendo muy populares últimamente. Hay muchos marcos de trabajo que soportan el diseño basado en componentes en diferentes lenguajes, algunos incluso son neutrales con respecto al lenguaje. Ejemplos de esto son el COM de Microsoft y el XPCOM de Mozilla.

La Arquitectura de Componentes de Zope (ZCA) es un marco de trabajo en Python que soporta el diseño y la programación basada en componentes. La ZCA funciona muy bien al desarrollar sistemas de software grandes en Python. La ZCA no es específica al servidor de aplicaciones Zope, se puede utilizar para desarrollar cualquier aplicación Python. Quizás debería llamarse la *Arquitectura de Componentes de Python.*

Hay dos paquetes principales relacionados con la arquitectura de componentes de Zope:

- `zope.interface` utilizado para definir la interfaz de un componente.
- `zope.component` se encarga de registrar y recuperar componentes.

El objetivo fundamental de la arquitectura de componentes de Zope es utilizar objetos Python de forma eficiente. Los componentes son objetos reusables con introspección para sus interfaces. Un componente provee una interfaz implementada en una clase, o cualquier objeto llamable. No importa cómo se implemente el componente, lo que importa es que cumpla los contratos definidos en su interfaz. Utilizando la arquitectura de componentes de Zope puedes distribuir la complejidad de sistemas entre varios componentes cooperantes. La arquitectura de componentes de Zope te ayuda a crear dos tipos básicos de componentes: *adaptador* y *utilidad.*

Recuerda, la ZCA no trata sobre los componentes en sí mismo, sino sobre la creación, registro y recuperación de los componentes. Recuerda también, un *adaptador* es una clase Python normal (o una fábrica en general) y una *utilidad* es un objeto llamable Python normal.

El marco de trabajo de la ZCA se desarrolla como parte del proyecto Zope 3. La ZCA, como ya se ha mencionado, es un marco de trabajo puramente Python, por tanto se puede utilizar en cualquier tipo de aplicación Python. Actualmente ambos proyectos Zope 3 y Zope 2 utilizan este marco de trabajo extensívamente. Hay otros muchos proyectos incluyendo aplicaciones no web que utilizan la Arquitectura de Componentes de Zope[1].

## 1.2 Una breve historia

El proyecto del marco de trabajo ZCA comenzó en 2001 como parte del proyecto Zope 3. Fue tomando forma a partir de las lecciones aprendidas al desarrollar sistemas software grandes usando Zope 2. Jim Fulton fue el

jefe de proyecto de este proyecto. Mucha gente contribuyó al diseño y a la implementación, incluyendo pero sin limitarse a, Stephan Richter, Philipp von Weitershausen, Guido van Rossum (también conocido como *Python BDFL*), Tres Seaver, Phillip J Eby y Martijn Faassen.

Inicialmente la ZCA definía componentes adicionales; *servicios* y *vistas*, pero los desarrolladores se dieron cuenta de que la utilidad podía sustituir *servicio* y el multi-adaptador podía sustituir *view*. Ahora la ZCA tiene un número muy pequeño de tipos de componentes principales: utilidades, adaptadores, subscriptores y manejadores. En realidad, subscriptores y manejadores son dos tipos especiales de adaptadores.

Durante el ciclo de la versión Zope 3.2, Jim Fulton propuso una gran simplificación de la ZCA[2]. Con esta simplificación se creó una nueva interfaz única (*IComponentRegistry*) para registrar componentes locales y globales.

El paquete `zope.component` tenía una larga lista de dependencias, muchas de las cuales no eran necesarias para una aplicación no Zope 3. Durante la PyCon 2007, Jim Fulton añadió la característica `extras_require` de setuptools para permitir la separación de la funcionalidad básica de la ZCA de las características adicionales[3].

Hoy el proyecto de la ZCA es un proyecto independiente con su propio ciclo de versiones y su repositorio Subversion. Sin embargo, los problemas y los errores aún se controlan como parte del proyecto Zope 3[4], y la lista principal zope-dev se utiliza para los debates de desarrollo[5].

## 1.3 Instalación

El paquete `zope.component`, junto con el paquete `zope.interface` son el núcleo de la arquitectura de componentes Zope. Ofrecen facilidades para definir, registrar y buscar componentes. El paquete `zope.component` y sus dependencias están disponibles en formato egg (huevo) desde el Índice de Paquetes Python (PyPI)[6].

Puedes instalar `zope.component` y sus dependencias utilizando *easy_install*[7]

```
$ easy_install zope.component
```

Este comando descargará `zope.component` y sus dependencias desde PyPI y los instalará en tu ruta Python.

Alternativamente, puedes descargar `zope.component` y sus dependencias desde PyPI y luego instalarlos. Instala los paquetes en el siguiente orden. En Windows, puede que necesitas los paquetes binarios de `zope.interface` y `zope.proxy`.

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

Para instalar estos paquetes, después de haberlos descargados, puedes utilizar el comando `easy_install` con los huevos como argumento. (También puedes darle todos estos huevos como argumneto en la misma linea.):

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

Estos métodos instalan la ZCA en el *Python de sistema*, en el directorio `site-packages`, lo cual puede causar problemas. En un mensaje a la lista de correo, Jim Fulton recomienda que no se utilice el Python de sistema[8].

## 1.4 Experimentando con código

`virtualenv` ay `zc.buildout` son herramientas que instalan la ZCA en un entorno de trabajo aislado. Esto es una buena práctica para experimentar con código y el estar familiarizado con estas herramientas será beneficioso para desarrollar e implantar aplicaciones.

Puedes instalar `virtualenv` usando `easy_install`:

```
$ easy_install virtualenv
```

Ahora crea un nuevo entorno así:

```
$ virtualenv miev
```

Esto creará un nuevo entorno virtual en el directorio `miev`. Ahora, desde dentro del directorio `miev`, puedes instalar `zope.component` y sus dependencias utilizando el `easy_install` que hay dentro del directorio `miev/bin`:

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

Ahora puedes importar `zope.interface` y `zope.component` desde el nuevo intérprete `python` dentro del directorio `miev/bin`:

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

Este comando ejecutará un intérprete de Python que puedes usar para ejecutar el código de este libro.

Utilizando `zc.buildout` con la receta `zc.recipe.egg` se puede crear un intérprete de Python con los huevos Python especificados. Primero instala `zc.buildout` usando el comando `easy_install`. (Puedes hacerlo también dentro de un entorno virtual). Para crear un nuevo buildout para experimentar con huevos Python, primero crea un directorio e inicialízalo usando el comando `buildout init`:

```
$ mkdir mibuildout
$ cd mibuildout
$ buildout init
```

Ahora el nuevo directorio `mibuildout` es un buildout. El archivo de configuración predeterminado de buildout es *buildout.cfg* . Después de la inicialización, tendrá el siguiente contenido:

```
[buildout]
parts =
```

Puedes cambiarlo a:

```
[buildout]
parts = py

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

Ahora ejecuta el comando `buildout` disponible dentro del directorio `mibuildout/bin` sin ningún argumento. Esto creará un nuevo intérprete Python dentro del directorio `mibuildout/bin`:

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

Este comando ejecutará un intérprete de Python que puedes usar para ejecutar el código de este libro.

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

# Chapter 2

# Un ejemplo

## 2.1 Introducción

Considera una aplicación de gestión para registrar los huéspedes que se hospedan en un hotel. Python puede implementar esto de varias formas distintas. Empezaremos con un mirada breve a un enfoque procedural, y después cambiaremos a un enfoque orientado a objetos básico. Mientras examinamos el enfoque orientado a objetos, veremos como como podemos beneficiarnos de los patrones de diseño clásicos, *adaptador* e *interface*. Esto nos llevará al mundo de la Arquitectura de Componentes de Zope.

## 2.2 Enfoque procedural

En una aplicación de gestión, el almacenamiento de los datos es muy importante. Por simplicidad, este ejemplo utilizará un diccionario Python como almacenamiento. Las claves del diccionario serán identificadores únicos para un huesped en particular. Y el valor será otro diccionario cuyas claves son los nombres de las propiedades:

```
>>> huespedes_db = {} #clave: id único, valor: detalles en un diccionario
```

En un método simplista, una función que acepte detalles como argumentos es suficiente para hacer el registro. También necesitas una función auxiliar para obtener el próximo identificador de tu almacenamiento de datos.

Esta función auxiliar, para obtener el próximo identificador se puede implementar así:

```
>>> def proximo_id():
...     claves = huespedes_db.keys()
...     if claves == []:
...         proximo = 1
...     else:
...         proximo = max(claves) + 1
...     return proximo
```

Como puedes ver, la implementación de la función *proximo_id* es muy simple. Bueno, no es la forma ideal, pero es suficiente para explicar conceptos. La función primero obtiene todas las claves del almacenamiento en una lista y comprueba si está vacía o no. Si está vacía, por tanto ningún elemento esta almacenado, devuelve *1* como el próximo identificador. Y si la lista no está vacía, el próximo identificador se calcula sumando *1* al valor máximo de la lista.

La función para registrar un huesped puede obtener el próximo identificador usando la función *proximo_id*, y luego asignando los detalles de un huesped usando un diccionario. Aquí está la función para obtener los detalles y almacenar en la base de datos:

```
>>> def registrar_huesped(nombre, lugar):
...     huesped_id = proximo_id()
...     huespedes_db[huesped_id] = {
...     'nombre': nombre,
...     'lugar': lugar
...     }
```

Aqui termina nuestro enfoque procedural. Será mucho más fácil añadir funcionalidades necesarias como almacenamiento de datos, diseño flexible y código testeable usando objetos.

## 2.3 Enfoque orientado a objetos

En una metodología orientada a objetos, puedes pensar en un objeto registrador que se encargue del registro. Hay muchas ventajas para crear un objeto que se encargue del registro. La más importante es que la abstracción que ofrece el objeto registrador hace el código más fácil de entender. Ofrece una forma de agrupar funcionalidad relacionada, y puede ser ampliada con herencia. Cuando se añadan mejoras, como cancelación y actualización de reservas, el objeto registrador puede crecer para tenerlas, o puede delegarselas a otro objeto:

```
>>> class RegistradorHuesped(object):
...
...     def registrar(self, nombre, lugar):
...         huesped_id = proximo_id()
...         huespedes_db[huesped_id] = {
...         'nombre': nombre,
...         'lugar': lugar
...         }
```

En esta implementación, el objeto registrador (una instancia de la clase RegistradorHuesped) se encarga del registro. Con este diseño, un objeto registrador en concreto puede realizar numerosos registros.

Así es como puedes usar la implementación actual:

```
>>> registrador = RegistradorHuesped()
>>> registrador.registrar("Pepito", "Pérez")
```

Los cambios de requisitos son inevitables en un proyecto real. Considera este caso, después de algún tiempo, un nuevo requisito se presenta: los huespedes también deben dar el número de teléfono para que se les admita. Necesitarás cambiar la implementación del objeto registrador para ofrecer esto.

Puedes cumplir este requisito añadiendo un argumento al método *registrar* y usar ese argumento en el diccionario de valores. Aquí está la nueva implementación para este requisito:

```
>>> class RegistradorHuesped(object):
...
...     def registrar(self, nombre, lugar, telefono):
...         huesped_id = proximo_id()
...         huespedes_db[huesped_id] = {
...         'nombre': nombre,
```

```
...         'lugar': lugar,
...         'telefono': telefono
...         }
```

Además de migrar los datos al nuevo esquema, ahora tienes que cambiar la forma de usar *RegistradorHuesped* en todos sitios. Si puedes abstraer los detalles de un huesped en un objeto y usarlo en el registro, los cambios en el código se pueden minimizar. Si sigues este diseño, tienes que pasarle el objeto huesped a la función en lugar de más argumentos. La nueva implementación con el objeto huesped quedaría así:

```
>>> class RegistradorHuesped(object):
...
...     def registrar(self, huesped):
...         huesped_id = proximo_id()
...         huespedes_db[huesped_id] = {
...         'nombre': huesped.nombre,
...         'lugar': huesped.lugar,
...         'telefono': huesped.telefono
...         }
```

Bien, incluso con esta implementación tienes que cambiar código. El cambio de código con nuevos requisitos es inevitable, tu objetivo es poder minimizar los cambios y hacerlo mantenible.

> **Note**
>
> Debes tener el coraje de hacer cualquier cambio, grande o pequeño, en cualquier momento. Retroalimentación inmediata es la única forma de que tengas el coraje. El uso de los tests automáticos te dan la retroalimentación inmediata y por tanto el coraje para hacer cambios. Para más información sobre el tema, puedes leer el libro llamado *Extreme Programming Explained* de Kent Beck.

Al introducir el objeto huesped, te has ahorrado un poco de escritura. Más que eso, la abstracción del objeto invitado ha hecho tu sistema mucho más simple y fácil de entender. Cuanto mejor se entienda mejor se puede restructurar y por tanto mejor se mantiene el código.

## 2.4 El patrón adaptador

Como se ha dicho antes, en una aplicación real, el objeto registrador puede tener funcionalidades de cancelación y/o actualización. Supón que hay dos método más como, *cancelar_registro* y *actualizar_registro*. En el nuevo diseño deberás pasar el objeto huesped a ambos métodos. Puedes solucionar este problema guardando el objeto huesped como un atributo del objeto registrador.

Aquí tenemos la nueva implementación del objeto registrador que guarda el objeto huesped como un atributo:

```
>>> class RegistradorHuespedNG(object):
...
...     def __init__(self, huesped):
...         self.huesped = huesped
...
...     def registrar(self):
...         huesped= self.huesped
...         huesped_id = proximo_id()
...         huespedes_db[huesped_id] = {
```

```
...         'nombre': huesped.nombre,
...         'lugar': huesped.lugar,
...         'telefono': huesped.telefono
...         }
```

La solución a la que has llegado es un patrón de diseño común llamado, *Adaptador*. Con este diseño, ahora puedes añadir más métodos, es decir más funcionalidad, si se necesita.

En esta implementación, al crear la instancia tienes que pasarle el objeto invitado que tiene los valores como atributos. Ahora es necesario crear instancias separadas de *RegistradorHuespedNG* para cada objeto huesped.

Ahora retrocedamos y pensemos de otra forma. Supón que eres el creador de este software y se lo vendes a muchos hoteles. Considera el caso en el que tus clientes necesitan distintos almacenamientos. Por ejemplo, un registrador puede almacenar los detalles en una base de datos relacional y otro puede almacenarlos en la Base de datos orientada a Objetos de Zope (ZODB). Sería mejor si puedes sustituir el objeto registrador por otro que almacena los detalles de los huespedes de otra forma distinta. Por tanto, un mecanismo para cambiar la implementación basado en alguna configuración será útil.

La arquitectura de componentes Zope ofrece un mecanismo para sustituir componentes basado en configuración. Utilizando la arquitectura de componentes de Zope puedes registrar componentes en un registro llamado registro de componentes. Después, puede recuperar componentes basandose en la configuración.

La clase *RegistradorHuespedNG* sigue, como ya has visto, un patrón llamado *Adaptador*. El *RegistradorHuespedNG* es el adaptador que adapta el objeto huesped (adaptado). Como puedes ver, el adaptador debe contener el objeto que adapta (adaptado). Esta es una implementación típica de un adaptador:

```
>>> class Adaptador(object):
...
...     def __init__(self, adaptado):
...         self.adaptado = adaptado
```

Ahora el adaptador puede usar el adaptado (llamar a sus métodos o acceder a sus atributos). Un adaptador puede adaptar más de un componente. La arquitectura de componentes zope ofrece un mecanismo para utilizar de forma efectiva este tipo de objetos. Así, qué componente se use se convierte en un problema de configuración.

Este es un escenario común donde quieres usar objetos diferentes para hacer las mismas cosas, pero los detalles varían. Hay muchas situaciones en programación donde quieres usar diferentes implementaciones para el mismo tipo de objetos. Te ofrecemos una pequeña lista de otros escenarios comunes:

- Un motor wiki que soporte múltiples marcados (STX, reST, Texto plano, etc.)
- Un objeto navegador que muestre el tamaño de distintos tipos de objetos.
- Diferentes tipos de formatos de salida para datos de texto (PDF, HTML etc.)
- Cuando se desarrolla una aplicación para múltiples clientes, sus requisitos pueden cambiar. Mantener distintas versiones del código de la misma aplicación para distintos clientes es difícil. Un enfoque mejor sería crear distintos componentes reutilizables y configurarlos basandose en los requisitos específicos de cada cliente.

Todos estos ejemplos señalan situaciones donde quieres hacer aplicaciones extensibles o enchufables. No utilices componentes *adaptadores* cuando no quieras extensibilidad o enchufabilidad.

La arquitectura de componentes de Zope ofrece componentes *adaptadores* para solucionar este tipo de problemas. De hecho, *RegistradorHuespedNG* es un adaptador sin declaración de interfaz explícita. Este tutorial tratará los adaptadores después de introducir el concepto de interfaces. Las interfaces son una de las bases de los componentes de Zope, por tanto entender el concepto y uso de interfaces es muy importante.

# Chapter 3

# Interfaces

## 3.1 Introducción

*Patrones de Diseño* es un libro clásico de ingeniería del software escrito por la *Banda de los Cuatro*[9]. En este libro se recomienda: "Programa contra un interfaz, no contra una implementación". Definir interfaces formales te ayuda a entender mejor el sistema. Además, las interfaces traen consigo todos los beneficios de la ZCA.

Las interfaces definen el comportamiento y el estado de objetos. Una interfaz describe como se trabaja con el objeto. Si te gustan las metaforas, piensa en la interfaz como un *contrato del objeto*. Otra métoda que ayuda es *molde de objetos*. En el código, los métodos y los atributos forman la interfaz del objeto.

La noción de interfaz es muy explícita en lenguajes modernos como Java, C#, VB.NET etc. Estos lenguajes también ofrecen una sintaxis para definir interfaces. Python tiene la noción de interfaces, per no es muy explícita. Para simular una definición formal de interfaces en C++, la *Banda de los Cuatro* utiliza clases con funciones virtuales en el libro *Patrones de Diseño*. De forma similar, la arquitectura de componentes de Zope utiliza la meta-clase heredada de `zope.interface.Interface` para definir una interfaz.

La base de la orientación a objetos es la comunicación entre los objetos. Se utilizan mensajes para comunicación entre objetos. En Python, funciones, métodos o cualquier otro llamable, puede usarse para manipular mensajes.

Por ejemplo, considera esta clase:

```
>>> class Anfitrion(object):
...
...     def buenosdias(self, nombre):
...         """Le dice buenos dias a los huespedes"""
...
...         return "¡Buenos días, %s!" % nombre
```

En la clase anterior, has definido un método *buenosdias*. Si llamas al método *buenosdias* desde un objeto creado con esta clase, devolverá *¡Buenos días, ...!*:

```
>>> anfitrion = Anfitrion()
>>> anfitrion.buenosdias('Pepe')
'¡Buenos días, Pepe!'
```

Aquí `anfitrion` es el objeto real. Los detalles de implementación de este objeto es la clase `Anfitrion`. Ahora, cómo se sabe cómo es el objeto, es decir, cuáles son los métodos y los atributos del objeto. Para responder a esto, tienes que ir a los detalles de implementación (la clase `Anfitrion`) del objeto o bien necesitas una documentación externa de la API[10].

Puedes usar el paquete `zope.interface` para definir la interfaz de objetos. Para la clase anterior puedes especificar la interfaz así:

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

>>> class IAnfitrion(Interface):
...
...     def buenosdias(huesped):
...         """Le dice buenos dias al huesped"""
```

Como puedes ver, la interfaz se define usando la sentencia class de Python. Usamos (¿abusamos de?) la sentencia class de Python para definir interfaces. Para hacer que una clase sea una interfaz, debe heredar de `zope.interface.Interface`. El prefijo `I` de la interfaz es una convención.

## 3.2 Declarando interfaces

Ya has visto como declarar una interfaz usando `zope.interfaz` en la sección anterior. En esta sección se explicarán los conceptos en detalle.

Considera esta interfaz de ejemplo:

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

>>> class IAnfitrion(Interface):
...     """Un objeto anfitrion"""
...
...     nombre = Attribute("""Nombre del anfitrion""")
...
...     def buenosdias(huesped):
...         """Le dice buenos dias al huesped"""
```

La interfaz, `IAnfitrion` tiene dos atributos, `nombre` y `buenosdias`. Recuerda que, al menos en Python, los métodos también son atributos de clases. El atributo `nombre` se define utilizando la clase `zope.interface.Attribute`. Cuando añades el atributo `nombre` a la interfaz `IAnfitrion`, no especificas ningún valor inicial. El propósito de definir el atributo `nombre` aquí es meramente para indicar que cualquier implementación de esta interfaz tendrá una atributo llamado `nombre`. En este caso, ¡ni siquiera dices el tipo que el atributo tiene que tener! Puedes pasar una cadena de documentación como primer argumento a `Attribute`.

El otro atributo, `buenosdias` es un método definido usando una definición de función. Nótese que no hace falta `self` en las interfaces, porque `self` es un detalle de implementación de la clase. Por ejemplo, un módulo puede implementar esta interfaz. Si un módulo implementa esta interfaz, habrá un atributo `nombre` y una función `buenosdias` definida. Y la función `buenosdias` aceptará un argumento.

Ahora verás como conectar *interfaz-clase-objeto*. Así objeto es la cosa viva y coleante, objetos son instancias de clases. Y la interfaz es la definición real del objeto, por tanto las clases son sólo detalles de implementación. Es por esto por lo que debes programar contra una interfaz y no contra una implementación.

Ahora deberías familiarizarte con dos términos más para entender otros conceptos. El primero es *proveer'y el otro es 'implementar*- Los objetos proveen interfaces y las clases implementan interfaces. En otras palabras, objetos proveen las interfaces que sus clases implementan. En el ejemplo anterior `anfitrion` (objeto) provee `IAnfitrion` (interfaz) y `Anfitrion` (clase) implementa `IAnfitrion` (interfaz). Un objeto puede proveer más de una interfaz y tambíen una clase puede implementar más de una interfaz. Los objetos también pueden proveer intefaces directamente, además de lo que sus clases implementen.

**Note**

Las clases son los detalles de implementación de los objetos. En Python, las clases son objetos llamables, asi que por qué otros objetos llamables no pueden implementar una intefaz? Sí, es posible. Para cualquier *objeto llamable* puedes declarar que produce objetos que proveen algunas interfaces diciendo que el *objeto llamable* implementa las interfaces. Generalmente los *objetos llamables* son llamados *fábricas*. Como las funciones son objetos llamables, una función puede ser la *implementadora* de una intefaz.

## 3.3 Implementando interfaces

Para declarar que una clase implementa una intefaz en particular, utiliza la función `zope.interface.implements` dentro de la sentencia class.

Considera este ejemplo, aquí `Anfitrion` implementa `IAnfitrion`:

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

>>> class Anfitrion(object):
...
...     implements(IAnfitrion)
...
...     nombre = u''
...
...     def buenosdias(self, huesped):
...         """Le dice buenos dias al huesped"""
...
...         return "¡Buenos dias, %s!" % huesped
```

**Note**

Si te preguntas como funciona la función `implements`, consulta el mensaje del blog de James Henstridge (http://blogs.gnome.org/jamesh/2005/09/08/python-class-advisors/) . En la sección del adaptador, verás una función `adapts`, que funciona de forma similar.

Como `Anfitrion` implementa `IAnfitrion`, instancias de `Anfitrion` proveen `IAnfitrion`. Hay unos cuantos métodos de utilidad que introspeccionan las declaraciones. La declaración se puede escribir fuera de la clase también. Si no escribes `interface.implements(IAnfitrion)` en el ejemplo anterior, entonces después de la sentencia class, puedes escribir algo como:

```
>>> from zope.interface import classImplements
>>> classImplements(Anfitrion, IAnfitrion)
```

## 3.4 Ejemplo revisado

Ahora volvemos a la aplicación de ejemplo. Ahora veremos como definir la interfaz del objeto registrador:

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

>>> class IRegistrador(Interface):
...     """Un registrador registrará los detalles de un objeto"""
```

```
...
...     def registrar():
...         """Registrar detalles de un objeto"""
...
```

Aquí primero has importado la clase `Interface` del módulo `zope.interface`. Si defines una subclase de esta clase `Interface`, será una interface desde el punto de vista de la arquitectura de componentes de Zope. Una interfaz puede ser implementada, como ya has visto, en una clase o cualquier otro objeto llamable.

La interfaz registrador definida aquí es `IRegistrador`. La cadena de documentación del interfaz da una idea del objeto. Al definir un método en la interfaz, has creado un contrato para el componente, en el que dice que habrá un método con el mismo nombre disponible. En la definición del método en la interfaz, el primer argumento no debe ser *self*, porque una intefaz nunca será instanciada ni sus métodos serán llamados jamás. En vez de eso, la sentencia class de la interfaz meramente documenta qué métodos y atributos deben aparecer en cualquier clase normal que diga que la implementa, y el parámetro *self* es un detalle de implementación que no necesita ser documentado.

Como sabes, una interfaz puede también especificar atributos normales:

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

>>> class IHuesped(Interface):
...
...     nombre = Attribute("Nombre del huesped")
...     lugar = Attribute("Lugar del huesped")
```

En esta interfaz, el objeto huesped tiene dos atributos que se especifican con documentación. Una interfaz también puede especificar atributos y métodos juntos. Una interfaz puede ser implementada por una clase, un módulo o cualquier otro objeto. Por ejemplo una función puede crear dinámicamente el componente y devolverlo, en este caso la función es una implementadora de la interfaz.

Ahora ya sabes lo que es una interfaz y como definirla y usarla. En el próximo capítulo podrás ver como se usa una interfaz para definir un componente adaptador.

## 3.5 Interfaces de marcado

Una interfaz se puede usar para declarar que un objeto en particular pertenece a un tipo especial. Un interfaz sin ningún atributo o método se llama *interfaz de marcado*.

Aquí tenemos una *interfaz de marcado*:

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

>>> class IHuespedEspecial(Interface):
...     """Un huesped especial"""
```

Esta interfaz se puede usar para declarar que un objeto es un huesped especial.

## 3.6 Invariantes

A veces te piden usar alguna regla para tu componente que implica a uno o más atributos normales. A este tipo de reglas se les llama *invariantes*. Puedes usar `zope.interface.invariant` para establecer *invariantes* para tus objetos en sus interfaces.

Considera un ejemplo sencillo, hay un objeto *persona*. Una persona tiene los atributos *nombre*, *email* y *telefono*. ¿Cómo implementas una regla de validación que diga que o bien el email o bien el teléfono tienen que existir, pero no necesariamente los dos?

Lo primero es hacer un objeto llamable, bien una simple función o bien una instancia llamable de una clase como esto:

```
>>> def invariante_contactos(obj):
...
...     if not (obj.email or obj.telefono):
...         raise Exception(
...             "Al menos una información de contacto es obligatoria")
```

Ahora defines la interfaz del objeto *persona* de esta manera. Utiliza la función `zope.interface.invariant` para establecer la invariante:

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

>>> class IPersona(Interface):
...
...     nombre = Attribute("Nombre")
...     email = Attribute("Direccion de email")
...     telefono = Attribute("Numero de telefono")
...
...     invariant(invariante_contactos)
```

Ahora usas el método *validateInvariants* de la interfaz para validar:

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

>>> class Persona(object):
...     implements(IPersona)
...
...     nombre = None
...     email = None
...     telefono = None

>>> pepe = Persona()
>>> pepe.email = u"pepe@algun.sitio.com"
>>> IPersona.validateInvariants(pepe)
>>> maria = Persona()
>>> IPersona.validateInvariants(maria)
Traceback (most recent call last):
...
Exception: Al menos una información de contacto es obligatoria
```

Como puedes ver el objeto *pepe* validó sin lanzar ninguna excepción. Pero el objeto *maria* no validó la restricción de la invariante, por lo que se lanzó la excepción.

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

# Chapter 4

# Adaptadores

## 4.1 Implementación

Esta sección describirá los adaptadores en detalle. La arquitectura de componentes de Zope, como has notado, ayuda a usar objetos Python de forma efectiva. Los adaptadores son uno de los componentes básicos usados por la arquitectura de componentes de Zope para usar los objetos Python de forma efectiva. Los adaptadores son objetos Python, pero con una interfaz bien definida.

Para decir que una clase es un adaptador utiliza la función *adapts' definida en el paquete "zope.component'*. Aquí tenemos un nuevo adaptador *RegistradorHuespedNG* con una declaración de interfaz explícita:

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

>>> class RegistradorHuespedNG(object):
...
...     implements(IRegistrador)
...     adapts(IHuesped)
...
...     def __init__(self, huesped):
...         self.huesped = huesped
...
...     def registar(self):
...         huesped = self.huesped
...         huesped_id = proximo_id()
...         huespedes_db[huesped_id] = {
...         'nombre': huesped.nombre,
...         'lugar': huesped.lugar,
...         'telefono': huesped.telefono
...         }
```

Lo que has definido aquí es un *adaptador* para *IRegistrador*, que adapta el objeto *IHuesped*. La interfaz *IRegistrador* es implementada por la clase *RegistradorHuespedNG*. Así, una instancia de esta clase proveerá la interfaz *IRegistrador*.

```
>>> class Huesped(object):
...
...     implements(IHuesped)
```

```
...
...     def __init__(self, nombre, lugar, telefono):
...         self.nombre = nombre
...         self.lugar = lugar
...         self.telefono = telefono

>>> pepe = Huesped("Pepito", "Perez")
>>> registrador_pepe = RegistradorHuespedNG(pepe)

>>> IRegistrador.providedBy(registrador_pepe)
True
```

'RegistradorHuespedNG' es simplemente un adaptador que has creado, puedes crear otros adaptadores que se encarguen del registro de huespedes de forma diferente.

## 4.2 Registro

Para usar este componente adaptador, tienes que registrarlo en un registro de componentes también conocido como administrador de sitios (site manager). Un administrador de sitios normalmente reside en un sitio. Un sitio y un administrador de sitios cobrarán más importancia al desarrollar una aplicación Zope 3. Por ahora sólo necesitas molestarte con el sitio global y el administrador de sitios global (o registro de componentes). Un administrador de sitios globales permanecerá en memoria, mientras que un administrador de sitios locales es persistente.

Para registrar tu componente, primero obtienes el administrador de sitios global:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(RegistradorHuespedNG,
...                     (IHuesped,), IRegistrador, 'ng')
```

Para obtener el administrador de sitios global, tienes que llamar a la función `getGlobalSiteManager` disponible en el paquete `zope.component`. En realidad, el administrador de sitios global está disponible como un atributo (`globalSiteManager`) del paquete `zope.component`. Asi que puedes usar directamente el atributo `zope.component.globalSiteManager`. Para registrar el adaptador, como ves más arriba, utiliza el método `registerAdapter` del registro de componentes. El primer argumento debe ser la clase/fábrica de tu adaptador. El segundo argumento es una tupla de objetos *adaptados*, i.e. el objeto que estás adaptando. En este ejemplo, estás adaptando sólo el objeto *IHuesped*. El tercer argumento es la interfaz que provista por el componente adaptador. El cuarto argumento es opcional, es el nombre de este adaptador en particular. Al darle un nombre a este adaptador, es un *adaptador con nombre*. Si no se le da un nombre, de forma predeterminada será la cadena vacía (").

En el registro anterior, le has dicho la interfaz adaptada y la interfaz a proveer por el adaptador. Como ya le habías dicho estos detalles en la implementación del adaptador, no es necesario que se especifique de nuevo. De hecho, podrías haber hecho el registro así:

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

Hay unas API antiguas para hacer el registro, que debes evitar. Las funciones antiguas empiezan por *provide*, por ejemplo, `provideAdapter`, `provideUtility`, etc. Cuando desarrollas una aplicación Zope 3 puedes usar el lenguaje de marcas de configuración de Zope (ZCML) para registrar componentes. En Zope 3, los componentes locales (componente persistentes) pueden registrarse desde la Interfaz de Administración de Zope (ZMI) o también pueden registrarse programáticamente.

Has registrado *RegistradorHuespedNG* con el nombre *ng*. De forma similar puedes registrar otros adaptadores con nombres distintos. Si un componente se registra sin nombre, de forma predeterminada será la cadena vacía.

> **Note**
>
> Los componentes locales son componentes persistentes pero los componentes globales están en memoria. Los componentes globales serán registrados basandose en la configuración de la aplicación. Los componentes locales se llevan a memoria desde la base de datos en el inicio de la aplicación.

## 4.3 Patrón de consulta

Recuperar componentes registrados a partir del registro de componentes se consigue mediante dos funciones disponibles en el paquete `zope.component`. Una de estas es `getAdapter` y la otra es `queryAdapter`. Las dos funciones aceptan los mismos argumentos. `getAdapter` lanzará `ComponentLookupError` si la búsqueda del componente falla mientras que `queryAdapter` devolverá *None*-

Puedes importar estos métodos así:

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

En la sección anterior has registrado un componente para el objeto huesped (adaptado) que provee la interfaz *IRegistrador* con el nombre 'ng'. En la primera sección de este capítulo, has creado un objeto huesped llamado *pepe*.

Así es como puedes recuperar un compomente que adapta la interfaz del objeto *pepe* (*IHuesped*) y ofrece la interfaz *IRegistrador* con el nombre 'ng'. Aquí las dos funciones `getAdapter` y `queryAdapter` funcionan similarmente:

```
>>> getAdapter(pepe, IRegistrador, 'ng') #doctest: +ELLIPSIS
<RegistradorHuespedNG object at ...>
>>> queryAdapter(pepe, IRegistrador, 'ng') #doctest: +ELLIPSIS
<RegistradorHuespedNG object at ...>
```

Como puedes ver, el primer argumento debe ser el adaptado, después la interfaz que debe ser ofrecida por el componente y por último el nombre del componente adaptador.

Si intentas buscar el componente con un nombre distinto al usado en el regsitro pero para el mismo adaptado e interfaz, la búsqueda fallará. Así es como funcionan ambos métodos en este caso:

```
>>> getAdapter(pepe, IRegistrador, 'no-existe') #doctest: +ELLIPSIS
Traceback (most recent call last):
...
ComponentLookupError: ...
>>> reg = queryAdapter(pepe,
...           IRegistrador, 'no-existe') #doctest: +ELLIPSIS
>>> reg is None
True
```

Como ves, `getAdapter` lanzó una excepción `ComponentLookupError`, pero `queryAdapter` devolvió *None* cuando la búsqueda falló.

El tercer argumento, el nombre del registro, es opcional. Si no le das el tercer argumento se usará el valor predeterminado que es la cadena vacía (''). Como no hay ningún componente registrado con una cadena vacía `getAdapter` lanzará `ComponentLookupError`. De forma similar `queryAdapter` devolverá *None*. Mira tú mismo como funciona:

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

En esta sección has aprendido cómo registrar un adaptador sencillo y cómo recuperarlo del registro de componentes. Estos tipos de adaptador se llaman adaptadores simples, porque adaptan sólamente un adaptado. Si un adaptador adapta más de un adaptado, entonces se le llama multi adaptador.

## 4.4 Recuperar un adaptador usando una interfaz

Los adaptadores se pueden recuperar directamente usando interfaces, pero sólo funcionará para adaptadores simples sin nombre. El primer argumento es el adaptado y el segundo argumento es un argumento de palabra clave. Si la búsqueda del adaptador falla, el segundo argumento es devuelto.

```
>>> IRegistrador(pepe, alternate='salida-predeterminada')
'salida-predeterminada'
```

La palabra clave se puede omitir:

```
>>> IRegistrador(pepe, 'salida-predeterminada')
'salida-predeterminada'
```

Si no le damos el segundo argumento, lanzará *TypeError*:

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

Ahora registramos 'RegistradorHuespedNG' sin nombre:

```
>>> gsm.registerAdapter(RegistradorHuespedNG)
```

Ahora la recuperación del adaptador debería funcionar:

```
>>> IRegistrador(pepe, 'salida-predeterminada') #doctest: +ELLIPSIS
<RegistradorHuespedNG object at ...>
```

Para los casos simples, puedes usar la interfaz para obtener componentes adaptadores.

## 4.5 Patrón de Adaptador

El concepto de adaptador de la Arquitectura de Componentes de Zope y el clásico *patrón adaptador* descrito en el libro Patrones de Diseño son muy similares. Pero la intención del uso del adaptador de la ZCA es más amplio que el *patrón adaptador*. La intención del *patrón adaptador* es convertir la interfaz de una clase en otra interfaz que los clientes esperan. Esto permite que clases que no hubieran podido colaborar debido a interfaces incompatibles puedan hacerlo. Pero en la sección *motivación* del libro de Patrones de Diseño, la Banda de los Cuatro dice: "A veces el adaptador es responsable de funcionalidad que la clase adaptada no ofrece". Los adaptadores de la ZCA hacen más hincapie en añadir funcionalidades en vez de en crear un nuevo interfaz para el objeto adaptado. Los adaptadores de la ZCA dejan que las clases adaptadoras aumenten la funcionalidad añadiendo métodos. (Es interesante notar que *Adaptador* se llamaba *Mejora* en una étapa inicial del diseño de la ZCA).[11]

El párrafo anterior tiene una cita del libro de la Banda de los Cuatro, que termina con algo como " ... la clase adaptada no ofrece". Pero en la siguiente frase yo utilizo "objeto adaptado" en vez de "clase adaptada", porque la Banda de los Cuatro describe dos variaciones de adaptadores basadas en la implementación. La primera se llama *clase adaptadora* y la otra se llama *objeto adaptador*. Una clase adaptadora utiliza herencia múltiple para adaptar una interfaz a otra, mientras que un objeto adaptador se basa en la composición de objetos. El adaptador de la ZCA sigue el patrón de objeto adaptador, que usa delegación como un mecanismo de composición. El segundo principio de orientación a objetos de la Banda de los Cuatro se algo como: "Favorece la composición de objetos sobre la herencia de clases". Para más detalles sobre el tema, por favor lee el libro de Patrones de Diseño.

Los mayores atractivos del adaptador de la ZCA son la interfaz explícita para componentes y el registro de componentes. Los componentes adaptadores de la ZCA se registran en el registro de componentes y son recuperados, cuando son necesarios, por objetos clientes usando la interfaz y el nombre.

[11] Discusión sobre el cambio de nombre de *Mejora* a *Adaptador*: http://mail.zope.org/pipermail/zope3-dev/2001-December/000008.html

# Chapter 5

# Utilidad

## 5.1 Introducción

Ahora ya conoces el concepto de interfaz, adaptador y registro de componentes. A veces es útil registrar un objeto que no adapta nada. Conexión a base de datos, analizador de XML, objetos que devuelven identificadores únicos, etc. son ejemplos de este tipo de objetos. Este tipo de componentes ofrecidos por la arquitectura de componentes de Zope se llaman componentes `utilidad`.

Las utilidades son sólo objetos que ofrecen una interfaz y que son recuperados con una interfaz y un nombre. Este enfoque crea un registro global en el que las instancias se pueden registrar y recuperar desde distintas partes de tu aplicación, sin necesidad de pasar las instancias arriba y abajo como parámetros.

No necesitas registrar todas las instancias de componentes así. Registra sólo aquellos componentes que quieras hacer sustituibles.

## 5.2 Utilidad simple

Antes de implementar la utilidad, como siempre, define su interfaz. Aqui tenemos una interfaz *saludador*:

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

>>> class ISaludador(Interface):
...
...     def saluda(nombre):
...         "dice hola"
```

Aquí tenemos una posible implementación de la interfaz anterior:

```
>>> class Saludador(object):
...
...     implements(ISaludador)
...
...     def saluda(self, nombre):
...         return "Hola " + nombre
```

Puedes registrar una instancia de esta clase usando `registerUtility`:

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

>>> saludador = Saludador()
>>> gsm.registerUtility(saludador, ISaludador)
```

En este ejemplo has registrado la utilidad como proveedora de la interfaz *ISaludador*. Puedes recuperar la utilidad con *queryUtility* o *getUtility*:

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

>>> queryUtility(ISaludador).saluda('Antonio')
'Hola Antonio'

>>> getUtility(ISaludador).greet('Susana')
'Hola Susana'
```

Como puedes ver, los adaptadores normalmente son clases, pero las utilidades normalmente son instancias de clases.

## 5.3 Utilidad con nombre

Cuando registras un componente utilidad, como un adaptador, puedes usar un nombre.

Por ejemplo, considera esto:

```
>>> saludador = Saludador()
>>> gsm.registerUtility(saludador, ISaludador, 'nuevo')
```

En este ejemplo has registrado una utilidad que ofrece la interfaz *IGreeter* con un nombre. Puedes buscar la interfaz tanto con *queryUtility* como con *getUtility*:

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

>>> queryUtility(ISaludador, 'nuevo').saluda('Antonio')
'Hola Antonio'

>>> getUtility(ISaludador, 'nuevo').saluda('Antonio')
'Hola Antonio'
```

Como puedes ver, cuando preguntas tienes que usar el *nombre* como segundo argumento.

> **Note**
>
> LLamar a la función *getUtility* sin un nombre (segundo argumento) es equivalente a llamarla con una cadena vacía (") comos el nombre. Porque, el valor predeterminado para el segundo argumento (nombrado) es una cadena vacía. Entonces, el mecanismo de recuperación de componentes intentará encontrar el componente con un nombre de cadena vacía ("), y fallará. Cuando la recuperación de un componente falla se lanza la excepción *ComponentLookupError*. Recuérdalo, no devolverá algún componente aleatorio registrado con cualquier otro nombre.

## 5.4 Fábrica

Una `Fábrica` es un componente utilidad que ofrece la intefaz `IFactory`.

Para crear una fábrica, primero tienes que definir el interfaz del objeto:

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

>>> class IBaseDatos(Interface):
...
...     def getConexion():
...         """Devuelve el objeto conexion"""
```

Aquí tenemos una implementación de mentira de la interfaz *IBaseDatos*:

```
>>> class BDFalsa(object):
...
...     implements(IBaseDatos)
...
...     def getConexion(self):
...         return "conexion"
```

Puedes crear una fábrica usando `zope.component.factory.Factory`:

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

>>> fabrica = Factory(BDFalsa, 'BDFalsa')
```

Ahora puedes registrarla así:

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

>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(fabrica, IFactory, 'bdfalsa')
```

Para usar la fábrica, puedes hacer lo siguiente:

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

Hay un atajo para usar la fábrica:

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

# Chapter 6

# Adaptadores avanzados

## 6.1 Multi adaptador

Un adaptador simple normalmente adapta sólo un objeto, pero un adaptador puede adaptar más de un objeto. Si un adaptador adapta más de un objeto, se le llama multi-adaptador.

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

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

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

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

>>> class MiFuncionalidad(object):
...     implements(IFuncionalidad)
...     adapts(IAdaptadoUno, IAdaptadoDos)
...
...     def __init__(self, uno, dos):
...         self.uno = uno
...         self.dos = dos

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

>>> gsm.registerAdapter(MiFuncionalidad)

>>> class Uno(object):
...     implements(IAdaptadoUno)

>>> class Dos(object):
...     implements(IAdaptadoDos)
```

```
>>> uno = Uno()
>>> dos = Dos()

>>> from zope.component import getMultiAdapter

>>> getMultiAdapter((uno, dos), IFuncionalidad) #doctest: +ELLIPSIS
<MiFuncionalidad object at ...>

>>> mifuncionalidad = getMultiAdapter((uno, dos), IFuncionalidad)
>>> mifuncionalidad.uno #doctest: +ELLIPSIS
<Uno object at ...>
>>> mifuncionalidad.dos #doctest: +ELLIPSIS
<Dos object at ...>
```

## 6.2 Adaptador de subscripción

A diferencia de los adaptadores normales, los adaptadores de subscripción se usan cuando queremos todos los adaptadores que adaptan un objeto a una interfaz particular. El adaptador de subscripción también se conoce como *subscriptor*.

Imagina un problema de validación. Tenemos objetos y queremos comprobar si cumplen algún tipo de estándar. Definimos una interfaz de validación:

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

>>> class IValidador(Interface):
...
...     def valida(ob):
...         """Determina si el objeto es válido
...
...         Devuelve una cadena que describe un problema de validación.
...         Se devuelve la cadena vacía para indicar que el objeto
...         es válido.
...         """
```

Quizá tengamos documentos:

```
>>> class IDocumento(Interface):
...
...     resumen = Attribute("Resumen del documento")
...     cuerpo = Attribute("Cuerpo del documento")

>>> class Documento(object):
...
...     implements(IDocumento)
...
...     def __init__(self, resumen, cuerpo):
...         self.resumen, self.cuerpo = resumen, cuerpo
```

Ahora puede que queramos especificar varias regalas de validación para los documentos. Por ejemplo, podemos requerir que el resumen sean una sola linea:

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

>>> class ResumenLineaUnica:
...
...     adapts(IDocumento)
...     implements(IValidador)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def valida(self):
...         if '\n' in self.doc.resumen:
...             return 'El resumen debe tener sólo una linea'
...         else:
...             return ''
```

O podemos exigir que el cuerpo tenga al menos una longitud de 1000 caracteres:

```
>>> class LongitudAdecuada(object):
...
...     adapts(IDocumento)
...     implements(IValidador)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def valida(self):
...         if len(self.doc.cuerpo) < 1000:
...             return 'demasiado corto'
...         else:
...             return ''
```

Podemos registrar estas reglas como adaptadores de subscripción:

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

>>> gsm.registerSubscriptionAdapter(ResumenLineaUnica)
>>> gsm.registerSubscriptionAdapter(LongitudAdecuada)
```

Podemos usar los subscriptores para validar objetos:

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

>>> doc = Documento("Un\nDocumento", "bla")
>>> [adaptador.valida()
...  for adaptador in subscribers([doc], IValidador)
...  if adaptador.valida()]
['El resumen debe tener sólo una linea', 'demasiado corto']
```

```
>>> doc = Documento("Un\nDocumento", "bla" * 1000)
>>> [adaptador.valida()
...  for adaptador in subscribers([doc], IValidador)
...  if adaptador.valida()]
['El resumen debe tener sólo una linea']

>>> doc = Document("Un Documento", "bla")
>>> [adaptador.valida()
...  for adaptador in subscribers([doc], IValidador)
...  if adaptador.valida()]
['demasiado corto']
```

## 6.3 Manejador

Los manejadores son factorias de adaptadores de subscripción que no producen nada. Realizan todo su trabajo cuando son llamados. Los manejadores se utilizan normalmente para tratar eventos. Los manejadores también se conocen como subscriptores de eventos o incluso como adaptadores de subscripción de eventos.

Los subscriptores de eventos son distintos a los adaptadores de subscripción en cuanto a que el que llama a los subscriptores de eventos no espera interactuar con ellos de manera directa. Por ejemplo, un publicador de eventos no espera recibir ningún valor de retorno. Ya que los subscriptores no necesitan ofrecer una API a los que los llaman, es más natural definirlos con funciones, en vez de con clases. Por ejemplo, en un sistema de gestión documental, podemos querer registrar tiempos de creación para los documentos:

```
>>> import datetime

>>> def documentoCreado(evento):
...     evento.doc.creado = datetime.datetime.utcnow()
```

En este ejemplo, tenemos una función que recibe un evento y realiza un procesamiento. Realmente no devuelve nada. Este es un caso especial de un adaptador de subscripción que adapta un evento a nada. Todo el trabajo se realiza cuando la "fábrica" del adaptador es invocada. Llamamos "manejadores" a los subscriptores que no crean nada. Hay APIs especiales para registrarlos y llamarlos.

Para registrar el subscriptor anterior definimos un evento de creación de documento:

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

>>> class IDocumentoCreado(Interface):
...
...     doc = Attribute("El documento que se creó")

>>> class DocumentoCreado(object):
...
...     implements(IDocumentoCreado)
...
...     def __init__(self, doc):
...         self.doc = doc
```

También cambiamos nuestra definición del manejador a:

```
>>> def documentoCreado(evento):
...     evento.doc.creado = datetime.datetime.utcnow()

>>> from zope.component import adapter

>>> @adapter(IDocumentoCreado)
... def documentoCreado(evento):
...     evento.doc.creado = datetime.datetime.utcnow()
```

Esto marca al manejador como un adaptador para los eventos *IDocumentoCreado*.

Ahora registraremos el manejador:

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

>>> gsm.registerHandler(documentoCreado)
```

Ahora, podemos crear un evento y usar la función *handle* para llamar a los manejadores registrados para el evento:

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

>>> handle(DocumentoCreado(doc))
>>> doc.creado.__class__.__name__
'datetime'
```

# Chapter 7

# Uso de la ZCA en Zope

La Arquitectura de Componentes de Zope se usa tanto en Zope 3 como en Zope 2. En este capítulo veremos el uso de la ZCA en Zope.

## 7.1 ZCML

El Lenguaje de Marcado de Configuración de Zope (ZCML) es un sistema de configuración basado en XML para registrar componentes. Así, en lugar de usar la API Python para los registros, puedes usar ZCML. Pero para usar ZCML, desafortunadamente, necesitas instalar más paquetes de dependencias.

Para instalar estos paquetes:

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

Para registrar un adaptador:

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

<adapter
    factory=".empresa.SalarioEmpleado"
    provides=".interfaces.ISalario"
    for=".interfaces.IEmpleado"
    />
```

Los atributos *provides* y *for* son opcionales, siempre que los hayas declarado en la implementación:

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

<adapter
    factory=".empresa.SalarioEmpleado"
    />
```

Si quieres registrar el componente como un adaptador con nombre, puedes darle un atributo *name*:

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

<adapter
```

```
    factory=".empresa.SalarioEmpleado"
    name="salario"
    />
```

Las utilidades se registran de forma similar.

Para registrar una utilidad:

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

<utility
    component=".basedatos.conexion"
    provides=".interfaces.IConexion"
    />
```

El atributo *provides* es opcional, siempre que lo hayas declarado en la implementación:

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

<utility
    component=".basedatos.conexion"
    />
```

Si quieres registrar el componente como una utilidad con nombre, puedes darle un atributo *name*:

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

<utility
    component=".basedatos.conexion"
    name="Conexión de Base de Datos"
    />
```

En vez de usar directamente el componente, también puedes dar una fábrica:

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

<utility
    factory=".basedatos.Conexion"
    />
```

## 7.2 Redefiniciones

Cuando registras componentes usando la API de Python (métodos `register*`), el último componente registrado sustituirá el componente registrado anteriormente, si los dos se registran con los mismos tipos de argumentos. Por ejemplo, imagina este ejemplo:

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

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

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

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

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

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

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

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

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

>>> gsm.registerAdapter(AP)

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

Si registras otro adaptador, el ya hay será sustituido:

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

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

Pero cuando se registran componente usando ZCML, el segundo registro causará un error de conflicto. Esto es una ayuda para tí, si no, cabría la posibilidad de que sustituyas los registros por error. Esto podría hacer que sea dificil de encontrar errores en tu sistema. Así, usar ZCML es un avance para la aplicación.

A veces es necesario sustituir registros existentes. ZCML ofrece una directiva `includeOverrides` para esto. Usándola, puedes escribir sustituciones en un fichero aparte:

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

## 7.3 NameChooser (Elejidor de nombres)

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

Este es un adaptador para elejir un nombre único para un objeto dentro de un contenedor.

El registro del adaptador es así:

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

A partir del registro, puedes ver que el adaptado es un `IWriteContainer` y el adaptador ofrece `INameChooser`.

Este adaptador ofrece una funcionalidad muy útil para los programadores de Zope. Las principales implementaciones de `IWriteContainer` en Zope 3 son `zope.app.container.BTreeContainer` y `zope.app.folder.Folder`. Normalmente heredarás de estas implementaciones al crear tus propias clases de contenedores. Supón que no hay interfaz llamada `INameChooser` ni adaptador, entonces tendrás que implementar esta funcionalidad para cada implementación de forma separada.

## 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

# Referencia

## 8.1 Attribute

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

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

Example:

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

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

## 8.2 Declaration

Need not to use directly.

## 8.3 Interface

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

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

Example 1:

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

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

Example 2:

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

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

## 8.4 adapts

This function helps to declare adapter classes.

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

Example:

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

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

## 8.5 alsoProvides

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

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

Example:

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

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

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

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

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

You can test it like this:

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

## 8.6 classImplements

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

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

Example:

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

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

```
...
...     name = Attribute("Name of person")

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

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

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

You can test it like this:

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

## 8.7 classImplementsOnly

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

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

Example:

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

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

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

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

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

You can test it like this:

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

## 8.8 classProvides

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

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

Example:

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

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

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

You can test it like this:

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

## 8.9 ComponentLookupError

## 8.10 createObject

Create an object using a factory.

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

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

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

Example:

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

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

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

>>> from zope.component.factory import Factory

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

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

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

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

## 8.11 directlyProvidedBy

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

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

Example:

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

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

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

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

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

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

>>> from zope.interface import directlyProvidedBy

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

## 8.12 directlyProvides

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

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

Example:

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

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

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

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

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

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

>>> from zope.interface import directlyProvidedBy

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

>>> ISmartPerson in providedBy(jack)
True

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

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

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

## 8.13 getAdapter

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

- Location: `zope.interface`

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

Example:

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

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

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

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

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

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

>>> IRegistrar.providedBy(jack_registrar)
True

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

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

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

## 8.14 getAdapterInContext

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

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

Example:

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

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

>>> context = Context(sm)

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

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

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

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

```
...         }

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

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

>>> IRegistrar.providedBy(jack_registrar)
True

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

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

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

## 8.15 getAdapters

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

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

Example:

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

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

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

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

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

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

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

## 8.16 getAllUtilitiesRegisteredFor

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

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

Example:

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

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

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

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

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

>>> from zope.component import getAllUtilitiesRegisteredFor

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

## 8.17 getFactoriesFor

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

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

Example:

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

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

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

>>> from zope.component.factory import Factory

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

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

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

>>> from zope.component import getFactoriesFor

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

## 8.18 getFactoryInterfaces

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

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

Example:

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

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

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

>>> from zope.component.factory import Factory

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

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

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

>>> from zope.component import getFactoryInterfaces

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

## 8.19 getGlobalSiteManager

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

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

Example:

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

## 8.20 getMultiAdapter

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

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

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

Example:

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

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

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

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

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

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

>>> gsm.registerAdapter(MyFunctionality)

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

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

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

>>> from zope.component import getMultiAdapter

>>> getMultiAdapter((one,two), IFunctionality) #doctest: +ELLIPSIS
<MyFunctionality object at ...>

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

## 8.21 getSiteManager

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

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

Example 1:

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

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

>>> context = Context(sm)

>>> from zope.component import getSiteManager

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

Example 2:

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

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

## 8.22 getUtilitiesFor

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

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

Example:

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

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

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

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

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

>>> from zope.component import getUtilitiesFor

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

## 8.23 getUtility

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

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

Example:

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

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

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

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

>>> greet = Greeter()
```

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

>>> from zope.component import getUtility

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

## 8.24 handle

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

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

Example:

```
>>> import datetime

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

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

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

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


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

>>> from zope.component import adapter

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


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

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

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

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

## 8.25 implementedBy

Return the interfaces implemented for a class' instances.

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

Example 1:

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

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

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

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

Example 2:

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

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

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

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

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

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

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

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

## 8.26 implementer

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

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

Example:

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

>>> @implementer(IFoo)
... def foocreator():
...     foo = Foo()
...     return foo
>>> list(implementedBy(foocreator))
[<InterfaceClass __builtin__.IFoo>]
```

## 8.27 implements

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

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

Example:

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

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

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

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

You can test it like this:

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

## 8.28 implementsOnly

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

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

Example:

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

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

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

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

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

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

```
You can test it like this:

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

## 8.29 moduleProvides

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

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

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

## 8.30 noLongerProvides

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

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

Example:

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

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

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

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

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

```
>>> jack.college = "New College"
>>> directlyProvides(jack, IStudent)

You can test it like this:

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

## 8.31 provideAdapter

It is recommend to use registerAdapter .

## 8.32 provideHandler

It is recommend to use registerHandler .

## 8.33 provideSubscriptionAdapter

It is recommend to use registerSubscriptionAdapter .

## 8.34 provideUtility

It is recommend to use registerUtility .

## 8.35 providedBy

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

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

Example 1:

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

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

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

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

You can test it like this:

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

Example 2:

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

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

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

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

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

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

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

## 8.36 queryAdapter

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

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

Example:

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

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

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

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

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

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

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

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

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

## 8.37 queryAdapterInContext

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

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

Example:

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

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

>>> context = Context(sm)

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

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

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

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

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

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

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

>>> IRegistrar.providedBy(jack_registrar)
True

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

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

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

## 8.38 queryMultiAdapter

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

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

Example:

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

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

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

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

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

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

>>> gsm.registerAdapter(MyFunctionality)

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

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

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

>>> from zope.component import queryMultiAdapter

>>> getMultiAdapter((one,two), IFunctionality) #doctest: +ELLIPSIS
<MyFunctionality object at ...>

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

## 8.39 queryUtility

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

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

Example:

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

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

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

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

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

>>> from zope.component import queryUtility

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

## 8.40 registerAdapter

Register an adapter factory.

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

Example:

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

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

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

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

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

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

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

>>> IRegistrar.providedBy(jack_registrar)
True

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

You can test it like this:

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

## 8.41 registeredAdapters

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

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

Example:

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

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

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

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

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

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

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

>>> IRegistrar.providedBy(jack_registrar)
True

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


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

## 8.42 registeredHandlers

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

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

Example:

```
>>> import datetime

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

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

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

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


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

>>> from zope.component import adapter

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


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

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

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

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

## 8.43 registeredSubscriptionAdapters

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

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

Example:

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

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

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

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

>>> from zope.component import adapts

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

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

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

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

## 8.44 registeredUtilities

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

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

Example:

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

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

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

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

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

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

## 8.45 registerHandler

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

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

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

Example:

```
>>> import datetime

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

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

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

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


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

>>> from zope.component import adapter

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


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

>>> gsm.registerHandler(documentCreated)

>>> from zope.component import handle

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

## 8.46 registerSubscriptionAdapter

Register a subscriber factory.

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

Example:

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

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

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

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

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

>>> from zope.component import adapts

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

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

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

## 8.47 registerUtility

Register a utility.

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

Example:

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

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

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

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

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

## 8.48 subscribers

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

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

Example:

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

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

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

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

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

>>> class SingleLineSummary:
...     adapts(IDocument)
...     implements(IValidate)
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if '\n' in self.doc.summary:
...             return 'Summary should only have one line'
...         else:
...             return ''

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

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

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

>>> from zope.component import subscribers

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

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

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

```
['too short']
```

## 8.49 unregisterAdapter

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

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

Example:

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

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

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

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

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

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

```
>>> jack_registrar = GuestRegistrarNG(jack)

>>> IRegistrar.providedBy(jack_registrar)
True

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

You can test it like this:

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

Now unregister:

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

After unregistration:

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

## 8.50 unregisterHandler

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

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

Example:

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

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

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

>>> doc = Document("A\nDocument", "blah")
```

```
>>> class IDocumentAccessed(Interface):
...     doc = Attribute("The document that was accessed")

>>> class DocumentAccessed(object):
...     implements(IDocumentAccessed)
...
...     def __init__(self, doc):
...         self.doc = doc
...         self.doc.count = 0

>>> from zope.component import adapter

>>> @adapter(IDocumentAccessed)
... def documentAccessed(event):
...     event.doc.count = event.doc.count + 1

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

>>> gsm.registerHandler(documentAccessed)

>>> from zope.component import handle

>>> handle(DocumentAccessed(doc))
>>> doc.count
1

Now unregister:

>>> gsm.unregisterHandler(documentAccessed)
True

After unregistration:

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

## 8.51 unregisterSubscriptionAdapter

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

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

Example:

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

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

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

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

>>> from zope.component import adapts

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

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

>>> gsm.registerSubscriptionAdapter(AdequateLength)

>>> from zope.component import subscribers

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

Now unregister:

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

After unregistration:

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

## 8.52 unregisterUtility

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

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

Example:

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

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

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

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

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

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

Now unregister:

>>> gsm.unregisterUtility(greet)
True

After unregistration:
```

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