Министерство образования и науки Российской федерации
Иркутский национальный исследовательский технический университет

Учреждение Российской академии наук
«Институт динамики систем и теории управления им. В. М. Матросова
Сибирского отделения РАН»

Е. А. Черкашин, М. Байджу

Компонентное программирование в Python

Учебное пособие

Иркутск 2016

УДК 681.3.06 (075.8)
ББК 32.973-01я73
Ч-48

Рекомендовано к изданию редакционно-издательским советом ИРНИТУ

Рецензенты:
зав. лаб. 4.1. ИДСТУ СО РАН, канд. техн. наук *А. Е. Хмельнов*,
ст. преп. каф. ИС. ИМЭИ ИГУ, канд. физ.-мат. наук *И. А. Казаков*

Черкашин Е. А.

Ч-48 Компонентное программирование в Python : учеб. пособие / Е. А. Черкашин, М. Байджу – Иркутск : Изд-во ИРНИТУ, 2016. – 141 с.

ISBN 978-5-9624-XXX-X

В пособии представлены материалы лекций и задания лабораторных работ по курсу «Сетевые операционные системы» Института кибернетики им. Е. И. Попова Иркутского государственного технического университета, тема «Компонентное проектирование распределенных систем», направление подготовки магистра – 09.04.01. Задача курса состоит в развитии навыков проектирования сложных программных комплексов на основе формального описания его компонент, их регистрации в репозитории компонент, конфигурировании программной системы.

Предполагаемая аудитория пособия: студенты, магистранты, аспиранты и молодые специалисты инженерных специальностей, связанных с программированием распределенных систем, серверов, сетей распределенной обработки информации, прикладных пакетов. Материал пособия адаптируем к курсам «Технологии программирования», «Открытые системы».

УДК 681.3.06 (075.8)
ББК 32.973-01я73

ISBN 978-5-9624-XXXX-X

Оглавление

Глава 1. Введение

Разработка больших программных систем практически всегда является сложным и трудоемким занятием, где объектно-ориентированный подход к анализу, проектированию и реализации (программированию) достаточно хорошо себя зарекомендовал. Проектирование и разработка программного обеспечения, использующая компонентный подход, становятся все более и более популярным методом проектирования. Компонентный подход и unit-тестирование позволяет достаточно просто разрабатывать и поддерживать программные системы. Существует много различных программных инфраструктур и инструментариев разработчика (framework), реализующих компонентное проектирование для разных языков программирования и их средах, некоторые даже не зависят от какой-либо конкретной среды разработки. Примерами выступают COM, разработанный Microsoft, и XPCOM - Mozilla.

Компонентная архитектура Zope (Zope Component Architecture, ZCA) – это одна из таких инфраструктур, разработанная для среды программирования Python, реализующая компонентный подход к проектированию и программированию. Он позволяет проектировать большие программные системы в среде программирования Python. И делает это совсем не плохо. Компонентная архитектура ZCA не требует для своего использования сервера приложений Zope, ZCA может использоваться в разработке любого Python-приложения. Справедливо было бы назвать ZCA *компонентной архитектурой Python* (Python Component Architecture).

ZCA позволяет использовать объекты Python эффективно. *Компоненты* – это объекты, которые легко использовать повторно, а к интерфейсам этих компонент можно получить полный доступ во время исполнения программы (introspection). *Интерфейс* – это объект, который описывает как следует взаимодействовать с конкретной компонентой. Другими словами, компонента *обслуживает сервисы* (provides)[1], определенные в интерфейсе. Компонента *реализуется* (implemented) в виде класса (class) или другого исполняемого объекта (callable object). И не важно, как реализована эта компонента, а вот что важно – это то, что она соответствует *требованиям*, определенным в интерфейсе (interface contract). При помощи ZCA сложность системы распределяется между множеством кооперирующихся компонент. Механизм такой кооперации обеспечивается двумя базовыми разновидностями компонент в ZCA: *адаптером* (adapter) и *утилитой* (utility).

[1] Слово ”provides”в исходном английском варианте текста имеет несколько основных вариантов перевода: ”обеспечивает ”реализует ”обслуживает ”оснащает ”оборудует”и ”образует собой”. Слово ”реализует”уже зарезервировано для перевода слова ”implements”: текст программного кода языка программирования Python представляет собой реализацию алгоритма метода, заданного в интерфейсе. Слово ”обеспечивать”в данном контексте имеет

ZCA распределен между тремя базовыми пакетами:

- `zope.interface`, используемый для определения (задания) интерфейсов компонент.
- `zope.event`, поддерживающий базовый механизм порождения и обработки событий.
- `zope.component`, обеспечивающий порождение и регистрацию компонент, а также их предоставление приложению по запросу.

Обращаю внимание, что ZCA сам по себе не является компонентой и не состоит из них, ZCA скорее механизм и библиотека для создания, регистрации и обеспечения доступа из программного кода приложения к зарегистрированным компонентам. В продолжение к предыдущему замечанию, *Адаптер* – это, как правило, обычный класс Python или фабрика классов (factory), а *утилита* – это обычный запускаемый объект в среде исполнения Python.

Среда ZCA разрабатывается как одна из задач проекта Zope 3. Как сказано выше ZCA является чисто питоновским фреймворком, и может использоваться в любых приложениях Python. В настоящее время ZCA активно используется в проектах Zope 3, Zope 2 и Grok. Существуют и другие приложения, в т. ч. и неинтернет-приложения, использующие ZCA[2].

1.1. Краткая история проекта

Проект ZCA начался в 2001 году и являлся частью проекта Zope 3. Причиной появления Zope 3 послужил анализ опыта, полученного при разработке комплексных программных систем на основе Zope 2. Джим Фултон (Jim Fulton) стал лидером проекта. Сообщество разработчиков, внесших свой вклад в дизайн и реализацию ZCA, возглавляется известными инженерами: Стефаном Рихтером (Stephan Richter), Филиппом фон Вайтершаусеном (Philipp von Weitershausen), Гвидо ван Россумом (Guido van Rossum), известным также как *Python BDFL*, Тресом Сивером (Tres Seaver), Филиппом Дж Эби (Phillip J Eby) и Мартином Фаассеном (and Martijn Faassen).

слишком абстрактный смысл, включающий всю семантику, образуемыми представленными вариантами перевода. Поэтому более менее точным и подходящим для цели изложения материала в данной книге вариантом перевода ”provides”видится слово ”обслуживает”в предположении, что в объекте (экземпляре класса) исполняется программный код метода, которому передано управление через данный интерфейс. Варианты ”оснащает ”оборудует”относятся не ко времени исполнения программного кода, а представляют собой статический аспект исходного слова ”provides”.

[2] `http://wiki.zope.org/zope3/ComponentArchitecture`

В начале проекта Zope 3 в ZCA существовали дополнительные компоненты – *сервисы* (services) и *представления* (views), но в процессе разработки оказалось, что *утилиты* заменяют полностью сервисы, а *мультиадаптеры* – представления. В настоящее время ZCA включает в себя небольшой набор базовых компонент: *утилиты*, *адаптеры*, *подписчики* (subscribers) и *обработчики* (handlers). На самом деле *подписчики* и *обработчики* – это два особых вида адаптеров.

В процессе подготовки версии Zope 3.2 Джим Фултон предложил значительно упростить среду компонентного программирования ZCA[3]. В результате этого появился новый общий интерфейс *IComponentRegistry*, позволяющий регистрировать как локальные так и глобальные компоненты.

В результате от пакета `zope.component` перестал зависеть достаточно длинный список других пакетов, при этом оставшиеся зависимости не относились напрямую к разработке приложений Zope 3. На конференции PyCon 2007 Джим Фултон предложил включить в пакет *setuptools* функцию *extras_require*, чтобы усилить уровень изолированности базового набора функций ZCA от вспомогательных функций[4].

В марте 2009, Трес Сивер окончательно устранил зависимости пакета от `zope.deferredimport` и `zope.proxy`.

Теперь проект ZCA является независимым со своим собственным планом выпусков и репозиторием Subversion. Проект как и прежде является составляющей частью среды разработки интернет-приложений Zope[5]. При этом ошибки и недочеты, выявляемые в ZCA, все еще публикуются на сайте системы отслеживания ошибок проекта Zope 3[6]. Кроме того, основной список рассылки *zope-dev* все еще используется для информационного обмена разработчиками ZCA[7]. Существует также список рассылки, предназначенный для пользователей Zope 3, *zope3-users*, в котором можно получить ответы на вопросы о ZCA[8].

1.2. Установка

Пакеты `zope.component`, `zope.interface` и `zope.event` являются ядром Компонентной архитектуры Zope. Они содержат технические средства для определения, регистрации и поиска компонент. Пакет `zope.component` и

[3] `http://wiki.zope.org/zope3/LocalComponentManagementSimplification`

[4] `http://peak.telecommunity.com/DevCenter/setuptools#declaring-dependencies`

[5] `http://docs.zope.org/zopeframework/`

[6] `https://bugs.launchpad.net/zope3`

[7] `http://mail.zope.org/mailman/listinfo/zope-dev`

[8] `http://mail.zope.org/mailman/listinfo/zope3-users`

пакеты, зависящие от него, доступны в формате egg на сайте Python Package Index (PyPI)[9].

Проще всего устанавливать `zope.component` и его зависимые пакеты при помощи программы *easy_install*[10]

```
$ easy_install zope.component
```

Программ загружает `zope.component`, его зависимые пакеты с сайта PyPI и устанавливает все как библиотеки интерпретатора Python.

С другой стороны, можно загрузить `zope.component` и зависящие от него пакеты с сайта PyPI и, затем, самостоятельно установить их. Установку пакетов следует производить в порядке, описанном ниже. В операционной системе Windows необходимо загрузить двоичные сборки следующих пакетов.

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

Для установки загруженных пакетов можно, как и раньше, использовать команду `easy_install`, при этом передавая в качестве параметра устанавливаемые пакеты, аналогично двоичным egg-пакетам. Команда позволяет все пакеты указывать одновременно в одной командной строке.

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

Кроме того, все пакеты можно разархивировать и установить по одному, например:

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

В результате исполнения этих команд ZCA устанавливается как общесистемная библиотека Python в директорий `site-packages`, что, иногда, создает некоторые проблемы. Джим Фултон в списке рассылки Zope 3 не рекомендовал устанавливать пакеты в общесистемном директории Python[11]. Их

[9] Хранилище пакетов Python: `http://pypi.python.org/pypi`

[10] `http://peak.telecommunity.com/DevCenter/EasyInstall`

рекомендуется устанавливать в виртуальное окружение, создаваемое помощи программы `virtualenv`, или пакета `zc.buildout`. Такой подход предоставляет возможности для проведения экспериментов с пакетами Python. Кроме того, в виртуальном окружении удобно развертывать уже готовые приложения.

1.3. Подготовка виртуального окружения

Существует множество популярных подходов к настройке изолированного виртуального окружения для среды программирования Python, удобного для разработки приложений. Наиболее популярные – это пакет `virtualenv`, разработанный Яном Бикингом (Ian Biking) и пакет `zc.buildout`, разработанный Джимом Фултоном. Оба этих пакета можно использовать и одновременно.

`virtualenv`

Пакет `virtualenv` устанавливается при помощи команды `easy_install`:

```
$ easy_install virtualenv
```

Затем создается новое окружение:

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

Команда `easy_install` создает новое окружение в директории `myve`. Теперь в этот директорий можно установить `zope.component` и пакеты, которые от него зависят. Команду необходимо запускать с использованием префикса `myve/bin`

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

Теперь можно попробовать импортировать пакеты `zope.interface` и `zope.component` из командной строки интерпретатора `python`, который находится в директории `myve/bin`:

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

После запуска Python выведет на экран его стандартное приглашение для ввода команд и выражений. В командной строке будем выполнять программный код, приводимый в книге.

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

`zc.buildout`

При помощи пакета `zc.buildout` и *рецепта* (recipe) `zc.recipe.egg` создаются специализированные версии интерпретатора Python с встроенными egg-пакетами. Для этого необходимо установить пакет `zc.buildout` при помощи команды `easy_install`, имеет смысл установить этот пакет также внутри виртуального окружения.

Чтобы создать новое buildout-окружение для проведения экспериментов с egg-пакетами Python, сначала создайте для него новый директорий, и, затем, создайте в нем так называемое buildout-окружение при помощи команды `buildout init`:

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

Теперь директорий `mybuildout` – новое buildout-окружение. Настройки окружения по умолчанию хранятся в файле *buildout.cfg*. Сначала в нем содержится следующий текст:

```
[buildout]
chapters =
```

Надо внести в настройки следующие изменения:

```
[buildout]
chapters = py

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

Запуск команды `buildout` из директория `mybuildout/bin` (без аргументов) создаст копию интерпретатора Python в директории `mybuildout/bin`:

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

Последняя команда выведет на экран приглашение Python, в котором будем запускать приводимый далее в книге программный код.

Глава 2. Пример задачи

Рассмотрим приложение, предназначенное для регистрации постояльцев в гостинице. В среде Python его можно реализовать множеством способов. Сначала кратко рассмотрим процедурный подход, затем перейдем к объектно-ориентированному программированию. После анализа результатов дизайна и реализации задачи при помощи объектно-ориентированного подхода, рассмотрим как данная задача решается при помощи классических шаблонов проектирования (паттернов проектирования, design patterns) *адаптер* и *интерфейс*. Таким образом у вас появится начальное представление о возможностях Компонентой архитектуры Zope.

2.1. Процедурный подход

Модуль хранения данных – очень важный момент в разработке приложений. В рассматриваемом примере будем использовать словарь (dictionary) языка Python в качестве такой системы хранения. Каждой записи соответствует сгенерированный уникальный ключ (ID). Ключ ассоциируется со значением, которое тоже будет словарем и будет содержать данные о бронировании комнаты в гостинице.

```
bookings_db = {} #key - уникальный идентификатор (ID), value -
↪ данные брони
```

В реализации системы нужна, как минимум, функция, которой передаются данные о бронировании, и функция, которая генерирует уникальное значение ID ключа для ассоциации с данными брони.

Сгенерировать значение ключа можно следующим образом:

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

Как видно из программы, реализация функции *get_next_id* достаточно проста. Функция получает из словаря список ключей. Если список пуст, то это значит, что произошло первое бронирование комнаты в гостинице, функция возвращает *1*. Если список не пуст, то добавить *1* к максимальному значению ключей и вернуть полученное значение.

Теперь созданную выше функцию встроим в программный код, создающий записи в словаре bookings_db:

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

В хранилище необходимо учесть, что может потребоваться сохранять и другие данные, такие как

- номер телефонов,
- требования к бронированию,
- способ платежа клиентом,
- ...

Необходимо реализовать следующие функции:

- отмена бронирования,
- уточнение бронирования,
- оплата проживания,
- постоянное хранение данных и доступ к ним,
- обеспечение защиты информации в хранилище,
- ...

Процедурная реализация данного примера требует создания большого числа функций, обменивающихся информацией друг с другом. Изменения требований к дизайну и реализации, добавление новых функций, создает условия к усложнению процесса поддержки программного кода. Кроме того, становится сложнее выявлять и исправлять ошибки.

На этом закончим обсуждение процедурного подхода. Обеспечивать постоянное хранение данных, гибкость дизайна и возможность тестирования программного кода, как правило, проще, используя объекты.

2.2. Объектно-ориентированный подход

system-message
WARNING/2 in izca-ru.txt, line 448
Explicit markup ends without a blank line; unexpected unindent. backrefs:

а) ”о создании объекта, обрабатывающего регистрацию”или же б) ”о создании класса, обрабатывающего регистрацию”?

В объектно-ориентированном подходе появляются *классы*, которые служат для инкапсуляции данных и программного кода, обрабатывающего эти данные в рамках одного абстрактного типа данных.

Основным классом в рассматриваемом примере является `FrontDesk`. Класс `FrontDesk`, а также другие классы, с которыми он взаимодействует, обеспечивают моделирование бизнес-процессов отеля и соответствующую обработку данных. При помощи *экземпляров* класса `FrontDesk` моделируется конкретный бизнес-процесс в конкретном отеле.

Как правило, консолидация программного кода и данных в объектах приводит в конечном счете к такому дизайну программной системы, который является более понятным, а программный кода проще тестировать и изменять.

Рассмотрим детали реализации класса `FrontDesk`:

```
class FrontDesk(object):

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

Далее, задачей объекта *frontdesk* (экземпляра класса *FrontDesk*) является обработка заявок на бронирование. Его используют следующим образом:

```
frontdesk = FrontDesk()
frontdesk.book_room(”Jack”, ”Bangalore”)
```

В реальности в каждый проект необходимо вносить изменения. Например, если руководство отеля решило, что каждый клиент должен при бронировании сообщать свой номер телефона, то программый код класса необходимо изменить.

Требование руководства можно удовлетворить добавив еще один аргумет к методу *book_room*, кроме того, новое ключевое значение добавляется к словарю:

```
class FrontDesk(object):

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

В результате изменилась и структура базы данных.

Теперь появилась необходимость во всей программе скорректировать вызов метода *book_room* – добавить новый аргумент *phone*. Если же инкапсулировать данные о постояльце в объекте *guest*, и передавать этот объект в качестве единственного аргумента функции регистрации, то указанные выше изменения будут минимизированы или вовсе не нужны. Изменения в объекте *guest*, например, добавление поля *phone* не вызовет необходимости в тотальной корректировки кода.

В результате получилось следующая реализация функции регистрации:

```
class FrontDesk(object):

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

Не смотря на полученный результат, программный код все-таки придется корректировать, чтобы отразить изменяющиеся во времени требования заказчика. Этого избежать практически никогда невозможно. Задача разработчика состоит в том, чтобы создать такой дизайн программы, который не требовал бы значительных изменений программного кода при изменении требований. Это свойство дизайна называется *Способность к сопровождению* (maintainability). Чем меньше нужно вносить изменений тем выше способность к сопровождению.

Note

В процессе реализации программы (кодирования) очень важно быть уверенным в том, что вносимые изменения не повлияют на работоспособность всего приложения. Существует способ обретения такой уверенности – автоматизированное тестирование. Если под рукой корректный и полный набор тестов вместе с системой контроля версий, то можно бесстрашно изменять программный код. Одним из рекомендуемых источников информации по этому вопросу является книга *Экстремальное программирование* Кента Бека**[#beck02]_**.

Итак, добавление объекта *guest* сократило программный код, и, что более важно, упростило программную систему. Программный код теперь проще понять, реструктурировать и поддерживать.

2.3. Шаблон проектирования *Адаптер*

В реальных приложениях объект *frontdesk* должен выполнять еще и другие функции, например, отмену и изменения бронирования. В текущей реализации необходимо передавать объект *guest* объекту *frontdesk* всякий раз при запуске методов *cancel_booking* и *update_booking*.

Можно избавиться от этого ограничения, если передать объект *guest* в конструктор FrontDesk.__init__(), сделав объект *guest* атрибутом нового экземпляра *FrontDesk*.

```
class FrontDeskNG(object):

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

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

    def cancel_booking(self):
        guest = self.guest
        # далее следует реализация отмены бронирования

    def update_booking(self):
```

```
        guest = self.guest
        # далее следует реализация изменения бронирования
```

Решение, к которому мы пришли, – это всем известный шаблон проектирования *Адаптер* (*Adapter*). Большинство адаптеров *содержат* в себе *адаптируемый объект* (*adaptee*):

```
class Adapter(object):

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

Этот шаблон чрезвычайно полезен в процессе реализации программного кода в следствии изменения

- требований заказчика к набору реализуемых функций программы;
- требований к системе хранения данных (ZODB, RDBM, XML ...);
- требований к выводу результата (HTML, PDF, plain text ...);
- способа представления исходного текста (ReST, Markdown, Textile ...).

В ZCA используются адаптеры, а также *регистраторы компонент*, которые позволяют настраивать реализацию блоков программного кода на основе *конфигураций* (*configuration*) компонентов.

Далее, в разделе об адаптерах ZCA, будет продемонстрированы преимущества использования конфигурирования, в частности,

- возможность переключения между вариантами реализации;
- возможность при необходимости добавлять новые реализации;
- повышается степень использования повторного кода: как унаследованного, так и программного кода ZCA.

Кроме того, использование конфигурирования позволяет создавать гибкий, масштабируемый программный код. Хотя, конечно, поддержка регистров компонент добавляет дополнительный уровень сложности приложения. Если же приложению в будущем не потребуются перечисленные возможности, то от ZCA будет мало пользы.

Теперь можно начать изучение Zope Component Architecture. Начнем с интерфейсов.

Глава 3. Интерфейсы

В файле README.txt[12] в path/to/zope/interface интерфейсы определяются следующим образом:

```
Интерфейсы~- это объекты, которые явным образом задают
(документируют) спецификацию внешнего поведения группы объектов.
Эти объекты, в свою очередь, ‘обслуживают‘ (provide) данные
интерфейсы.  Интерфейс задает спецификацию поведения при помощи

- Информативной документации в виде doc-строк питона;

- Определений атрибутов и их свойств;

- Инвариантов, т. е. условий, которые должны выполняться во всех
  объектах, обслуживающих данный интерфейс.
```

В классической книге по инженерии программного обеспечения *Приемы объектно-ориентированного проектирования. Паттерны проектирования*[13] известной *Банды Четырех* дана следующая рекомендация: ”Программируй, ориентируясь на интерфейс, а не на реализацию”. Формальная спецификация интерфейса формирует понимание системы в целом. Кроме того, интерфейсы позволяют использовать все преимущества, предоставляемые ZCA.

Интерфейс определяет (формально задает) все характеристики, поведение и предоставляемые возможности объекта. Интерфейсы описывают *что* объект может делать, но чтобы узнать *как* он это делает необходимо изучать его реализации.

Интерфейсам соответствуют синонимы *контракт* и *эскиз* (*чертеж*), которые задают правовые и архитектурные положения набора спецификаций.

В некоторых современных системах программирования, таких как Java, C#, VB.NET и т. п., интерфейсы являются синтаксическими элементами языка. В языке Python нет таких языковых конструкций, поэтому интерфейсы в ZCA реализованы при помощи специальных метаклассов. Интерфейс задается при помощи наследования этих метаклассов.

Рассмотри пример в классическом стиле *hello world*:

```
class Host(object):
```

[12] Дерево исходных кодов Zope изобилует текстовыми файлами README.txt, которые представляют собой достаточно полую документацию к сопровождаемым модулям

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

```
        def goodmorning(self, name):
            """Пожелаем доброго утра постояльцам"""

            return "Доброе утро, %s!" % name
```

В классе *Host* определен метод *goodmorning*. Если у объекта (экземпляра данного класса) вызвать метод *goodmorning* на экран выведется сообщение *Доброе утро, ...!*

```
    host = Host()
    host.goodmorning('Jack')
'Доброе утро, Jack!'
```

Здесь, `host` – это объект, который является частью исполняющегося приложения. Чтобы увидеть его детали реализации, необходимо получить доступ к классу `Host` либо через исходный код, либо при помощи инструментов документирования API[14] .

Самое время начать использовать интерфейсы ZCA. Для класса *Host* интерфейс задается следующим образом:

```
    from zope.interface import Interface

    class IHost(Interface):

        def goodmorning(guest):
            """Пожелаем доброго утра постояльцу"""
```

Как видно из примера, интерфейс *IHost* унаследован от zope.interface.Interface. На самом деле конструкция выглядит так же как как питоновский класс, но он задает интерфейс в рамках среды ZCA. Приставка `I` – это полезное соглашение о форме и способе задания имен интерфейсов.

3.1. Декларация интерфейса

В предыдущем разделе представлен пример декларации интерфейса при помощи `zope.interface`. Здесь рассмотрим основные правила спецификаций интерфейсов более подробно.

Рассмотрим следующий пример определения интерфейса:

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

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

```
class IHost(Interface):
    """Объект host"""

    name = Attribute("""Имя владельца отеля""")

    def goodmorning(guest):
        """Пожелаем доброго утра постояльцу"""
```

Интерфейс `IHost` включает два атрибута `name` и `goodmorning`. Надо сказать, что в языке Python методы также являются атрибутами класса. Атрибут `name` декларируется при помощи класса `zope.interface.Attribute`. Сама декларация атрибута `name` в интерфейсе `IHost` не устанавливает ему какого-либо начального значения. Идея состоит в том, чтобы указать, что все реализации интерфейса `IHost` будут так или иначе использовать этот атрибут. Более того, данная декларация никоим образом не задает тип атрибута! В качестве первого аргумента `Attribute` передается строка, документирующая описываемый атрибут.

Второй атрибут, `goodmorning`, является методом и задается синтаксически как функция. Заметим, что параметр *self* в методах, определяющих интерфейс, не используется, так как *self*, ссылка на экземпляр класса, – это деталь реализации класса. Теперь, если программный модуль реализует интерфейс `IHost`, то в его программном коде будут использованы атрибут `name` и метод `goodmorning`. Методу `goodmorning` передается ровно один параметр.

Теперь изучим как объединять тройку *интерфейс-класс-объект*. Объект – это реально функционирующая сущность, объекты являются экземплярами классов. Именно интерфейс определяет структуру и внешнее поведение объекта, а классы – это лишь детали реализации. Именно по этой причине программный код реализуется, ориентируясь на спецификацию интерфейса.

В излагаемой концепции присутствуют еще две терминологические единицы. Первая – *обслуживать* (*provide*), вторая – *реализовать* (*implement*). Объекты обслуживают интерфейсы, а классы реализуют интерфейсы. Другими словами, объекты обслуживают интерфейсы, которые их классы реализуют. В приведенном выше примере объект `host` обслуживает интерфейс `IHost`, класс `Host` реализует интерфейс `IHost`. Один объект может обслуживать несколько интерфейсов, также как класс может реализовывать несколько интерфейсов. Кроме того, в дополнение к реализациям интерфейса в классе, объекты могут обслуживать интерфейсы в обход класса, т. е., непосредственно (directly).

Note

Классы – это детали реализации объектов. В Python классы сами по себе являются исполняемыми (callable) объектами, так почему же не дать возможность другим исполняемым объектам реализовывать интерфейсы. Такая возможность существует. Для любого исполняемого объекта можно зарегистрировать в ZCA интерфейс и тот факт, что этот объект порождает объекты, обслуживающие данный интерфейс. Такие объекты, в общем случае, называются *фабриками*. Функции также являются исполняемыми объектами, т. е. они тоже могут реализовывать интерфейсы.

3.2. Реализация интерфейсов

Чтобы указать, что класс реализует некоторый интерфейс, используется функция `zope.interface.implements` в определении класса.

Рассмотрим пример, здесь класс `Host` реализует интерфейс `IHost`:

```
from zope.interface import implements

class Host(object):

    implements(IHost)

    name = u”

    def goodmorning(self, guest):
        ”””Пожелаем доброго утра постояльцу”””

        return ”Доброе утро, %s!” % guest
```

Note

Если вам интересно как работает функция `implements`, обратитесь в блог-пост Джеймса Хенстриджа (James Henstridge) (`http://blogs.gnome.org/jamesh/2005/09/08/python"=class-advisors/`) . В разделе *adapter* находится функция `adapts`, которая функционирует аналогично функции `implements`.

Так как класс `Host` реализует интерфейс `IHost`, то экземпляры класса `Host` обслуживают интерфейс `IHost`. В ZCA есть в ряд средств для интроспекции имеющихся деклараций интерфейсов. Кроме того, декларации могут быть сделаны вне определения класса: если не указывать `implements(IHost)` в приведенном выше примере, то после определения класса `Host` данную декларацию можно сделать следующим образом:

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

3.3. Компонентный подход в рассматриваемом примере

Итак, вернемся к нашему примеру приложения. Рассмотрим определение интерфейса объекта *стол регистрации* (frontdesk):

```
from zope.interface import Interface

class IDesk(Interface):
    """Стол регистрации, регистрирующий данные о постояльце"""

    def register():
        """Регистрируем данные постояльца"""
```

В начале импортируется класс `Interface` из модуля `zope.interface`. Каждый подкласс класса `Interface` с точки зрения компонентной архитектуры Zope является определением интерфейса. Интерфейсы реализуются при помощи классов или любых других исполняемых (callable) объектов языка Python.

В примере задан интерфейс стола регистрации – `IDesk`. Строка, которая находится сразу под определением интерфейса (строка документирования), объясняет предназначение интерфейса. Определение метода в интерфейсе задает контрактное обязательство компоненты – в ней будет обслуживаться одноименный метод. В определении метода в интерфейсе не надо указывать первый аргумент *self*, так как интерфейс не предназначен для порождения экземпляров, т. е. эти декларации методов никогда не будут выполнены. Главная задача интерфейса – спецификация перечня атрибутов и методов, которые должны быть определены классе, вязавшем на себя обязательства реализовать данный интерфейс. Поэтому параметр *self* – это деталь реализации метода в классе, а в интерфейсе его указывать нет необходимости.

Кроме методов ZCA разрешает в интерфейсе декларировать обычные атрибуты:

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

class IGuest(Interface):

    name = Attribute("Имя постояльца")
    place = Attribute("Место размещения постояльца")
```

Данный интерфейс сообщает, что соответствующие *guest*-объекты включают два атрибута. В ZCA в интерфейсах можно задавать одновременно и атрибуты и методы. Интерфейсы, как правило, реализуются при помощи классов и модулей, но можно и при помощи других объектов. Например, функция, которая динамически создает и возвращает компоненты, является таким нетипичным примером реализации интерфейса.

Итак, теперь вам известно, что такое интерфейс, как его задавать и использовать. В одном из следующих разделов рассмотрим использование интерфейсов в спецификации и реализации компонент-адаптеров.

3.4. Интерфейсы-маркеры

В некоторых случаях интерфейсы используются для указания, что некоторый объект принадлежит специальному типу или обладает некоторыми специфическими свойствами. Такие интерфейсы обычно не содержат определений атрибутов и методов и называется *маркерным интерфейсом* (*marker interface*).

Вот простой пример маркерного интерфейса:

```
from zope.interface import Interface

class ISpecialGuest(Interface):
    """Особый гость"""
```

Данный интерфейс указывает, что помеченный объект – это не просто *гость*, а *особый гость*.

3.5. Инварианты

Иногда бывает необходимо явным образом указать, что атрибуты компоненты связаны друг с другом некоторым логическим правилом или ограничением. Механизм задания таких правил и ограничений в ZCA реализуется при помощи *инвариантов* (*invariants*). Инварианты задаются также в интерфейсах при помощи функций и объектов модуля `zope.interface.invariant`.

Рассмотрим простой пример, пусть существует объект *person* (человек). Его можно описать при помощи атрибутов *имя* (*name*), *email* и *телефон* (*phone*). Теперь можно задать ограничение, которое требует, чтобы email или телефон были обязательно заданы у каждого объекта person, однако требовать, чтобы оба этих атрибута были заданы, мы не будем.

Ограничение проверяет специальный исполняемый объект. В качестве такого объекта выступает исполняемый экземпляр некоторого класса или даже обычная функция, например такая:

```
    def contacts_invariant(obj):

        if not (obj.email or obj.phone):
            raise Exception(
                "Необходимо указать как минимум один вид контакта")
```

Затем надо определить интерфейс объекта *person*, используя функцию `zope.interface.invariant`, предписывающую, что для компоненты задается инвариант:

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

    class IPerson(Interface):

        name = Attribute("Имя")
        email = Attribute("Email-адрес")
        phone = Attribute("Номер телефона")

        invariant(contacts_invariant)
```

Теперь при помощи метода *validateInvariants* можно проверять (верифицировать) структуру объекта на непротиворечивость содержащихся в нем данных:

```
    from zope.interface import implements

    class Person(object):
        implements(IPerson)

        name = None
        email = None
        phone = None

    jack = Person()
    jack.email = u"jack@some.address.com"
    IPerson.validateInvariants(jack)
    jill = Person()
    IPerson.validateInvariants(jill)
Traceback (most recent call last):

Exception: "Необходимо указать как минимум один вид контакта"
```

Из примера видно, что объект *jack* верифицирован успешно, и при этом не возникло никаких исключений. Объект *jill*, наоборот, не прошел проверку ограничения, что привело к возникновению соответствующего исключения.

Глава 4. Адаптеры

4.1. Реализация

Компонентная архитектора Zope, как уже неоднократно говорилось, направлена на продуктивное использование объектов Python. Компоненты-адаптеры – это одна из базовых концепций ZCA, реализуемая и используемая для повышения продуктивности процесса программирования. Как и прежде, адаптеры – это объекты Python, но со специальным образом оформленным интерфейсом.

Для того, чтобы указать, что класс является адаптером, используется функция *adapts*, которая находится в пакете `zope.component`. В следующем примере в тексте определения адаптера *FrontDeskNG* включена декларация интерфейса, который реализует класс, и декларация адаптируемого интерфейса:

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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

Итак, компонента *FrontDeskNG* – адаптер интерфейса *IGuest* к интерфейсу *IDesk*. Это значит, что везде, где требуются компоненты, обслуживающие интерфейс *IDesk*, можно использовать компоненты, обслуживающие *IGuest*, но через компоненту-адаптер *FrontDeskNG*. Интерфейс *IDesk* реализуется в классе *FrontDeskNG*, т. е., экземпляры этого класса обслуживают интерфейс *IDesk*.

```
class Guest(object):
```

```
        implements(IGuest)

        def __init__(self, name, place):
            self.name = name
            self.place = place

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

    IDesk.providedBy(jack_frontdesk)
True
```

Можно разработать и другие адаптеры, регистрирующие новых постояльцев каким-либо другим способом.

4.2. Регистрация адаптера

Для того, чтобы использовать компоненту-адаптер, необходимо ее сначала зарегистрировать в реестре компонент, который в ZCA называется менеджером сайта (site manager). Обычно реестр компонент ассоциируется с некоторым интернет-сайтом. Такое название компоненты-реестра унаследовано от программной системы Zope 3, являющейся средой для разработки динамических сайтов, основанной на ZCA. Сейчас важно только знать, что в приложении есть некий глобальный сайт и его глобальный менеджер сайта, выполняющий функции реестра. Глобальный менеджер сайта располагается в оперативной памяти, а локальные – в долговременной.

Для регистрации компоненты сначала надо получить ссылку на глобальный реестр.

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

Для того, чтобы найти глобальный реестр, необходимо вызвать функцию `getGlobalSiteManager`, которая находится в модуле `zope.component`. Фактически, реестр доступен как атрибут `globalSiteManager` пакета `zope.component`. То есть, можно просто использовать ссылку `zope.component.globalSiteManager`. Для регистрации адаптера используется метод `registerAdapter` реестра компонент. Первым аргументом метода должен быть либо класс, реализующий адаптер, либо какая-либо фабрика. Вторым аргументом выступает набор (tuple) адаптируемых интерфейсов компонент. В приведенном примере адаптируются только компоненты, обслуживающие интерфейс *IGuest*. Третьим аргу-

ментом является интерфейс, обслуживаемый компонентой-адаптером. Четвертый аргумент является необязательным, он позволяет идентифицировать адаптеры со специальными свойствами. В примере такой идентификатор как раз и был использован, в результате получился специальный поименованный адаптер (*named adapter*). Если название адаптеру не задано, то по умолчанию оно является пустой строкой (”).

В определении адаптера *FrontDeskNG* задаются и адаптируемый интерфейс и интерфейс, к которому происходит адаптация. То же самое задано и при регистрации адаптера, что бессмысленно дублирует информацию. Особой необходимости в этом нет, поэтому можно зарегистрировать адаптер *FrontDeskNG* следующим упрощенным вызовом:

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

В ZCA остался устаревший API регистрации, который не следует использовать. Функции устаревшего API начинаются с *provide*, например: `provideAdapter`, `provideUtility` и т. д. При разработке приложений Zope 3 достаточно удобно использовать еще один способ регистрации компонент – Язык разметки для конфигурирования Zope (Zope configuration markup language, ZCML). Локальные компоненты Zope 3, т. е. компоненты, которые сохраняются в долговременной памяти, можно также регистрировать из Интерфейса управления Zope (Zope Management Interface, ZMI), ну и программным способом регистрация тоже может быть выполнена.

Адаптер *FrontDeskNG* зарегистрирован под названием *ng*. Подобным же образом можно регистрировать другие адаптеры для этой же комбинации интерфейсов под другими именами. Напомним, что если компонента зарегистрирована без названия, то в качестве названия используется пустая строка.

Note

Локальные компоненты сохраняются в долговременной дисковой памяти, в отличие от глобальных, которые всегда хранятся в оперативной памяти. Глобальные компоненты, как правило, регистрируются в момент конфигурирования приложения. Локальные компоненты загружаются в оперативную память из базы данных при загрузке приложения.

4.3. Запрос адаптера в реестре

Доступ к зарегистрированной компоненте из реестра компонент осуществляется при помощи двух функций из пакета `zope.component`. Первая функция называется `getAdapter`, а другая – `queryAdapter`. Обе функции принимают одни и те же параметры и возвращают требуемую компоненту, если

таковая была зарегистрирована. Если поиск компоненты в реестре завершился неудачей, то `getAdapter` создает исключительную ситуацию `ComponentLookupError`, `queryAdapter` просто возвратит *None*.

Импорт этих функций выполняется следующим образом:

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

В следующем примере нам понадобиться поименованная компонента для адаптируемого объекта *guest*, который обслуживает интерфейс *IDesk*. Компонента реализована и зарегистрирована под именем 'ng' в предыдущем разделе. Объект *guest* с именем *jack* создан в первом разделе этой главы.

Пример демонстрирует, как можно получать доступ к поименованной ('ng') компоненте, адаптирующей интерфейс (*IGuest*) объекта *jack* и обслуживающей интерфейс *IDesk*. Здесь функции `getAdapter` и `queryAdapter` исполняются одинаково:

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

В вызове функции первый аргумент – адаптируемый объект, второй – интерфейс, к которому необходимо этот объект адаптировать. Последний аргумент функции – идентификатор адаптера. Если попытаться получить доступ с идентификатором, который не использовался ранее при регистрации адаптеров таких же объектов к такому же интерфейсу, то по понятным причинам это завершиться неудачей. Приведем пример использования функций и продемонстрируем различие в их работе:

```
    getAdapter(jack, IDesk, 'not-exists')
Traceback (most recent call last):

ComponentLookupError:
    reg = queryAdapter(jack,
               IDesk, 'not-exists')
    reg is None
True
```

Видно, что в результате неудачного доступа в `getAdapter` возникло исключение `ComponentLookupError`, а `queryAdapter` мирно вернул *None*. Если при получении доступа к компоненте не использовать ее идентификатор, это означает, что он будет равен пустой строке. И если компоненты без идентификатора не было зарегистрировано, то получить такой доступ тоже невозможно. Как и прежде в `getAdapter` возникнет исключение `ComponentLookupError`, а `queryAdapter` вернет *None*. Вот пример такого неудачного доступа:

```
    getAdapter(jack, IDesk)
Traceback (most recent call last):

ComponentLookupError:
    reg = queryAdapter(jack, IDesk)
    reg is None
True
```

В этом разделе продемонстрировано как регистрируются простые адаптеры, и как потом получать к ним доступ при помощи реестра компонент. Простые адаптеры называются единичными адаптерами, так как адаптируют только один объект. Если компонента адаптирует более одного объекта, то она называется мультиадаптером.

4.4. Получение доступа к адаптеру с использованием интерфейса

Получить доступ к адаптеру можно, используя только интерфейс. Этот метод работает только для непоименованных адаптеров, адаптирующих только один объект. Первый аргумент – адаптируемый объект, второй аргумент, ключевое слово, – объект, который возвращается, если доступ к компоненте оказывается неудачным.

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

Второй аргумент можно задавать и не в формате ключевого слова:

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

Если второй аргумент совсем не задан, то неудачная попытка доступа порождает исключение *TypeError*:

```
    IDesk(jack)
Traceback (most recent call last):

TypeError: ('Could not adapt',
  <Guest object at ... >,
  <InterfaceClass __builtin__.IDesk>)
```

Зарегистрируем *FrontDeskNG* как непоименованный адаптер:

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

Теперь доступ должен быть удачно получен:

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

В простых ситуациях можно спокойно использовать интерфейсы для получения доступа к компонентам-адаптерам.

4.5. Шаблон проектирования *Adapter*

Концепция использования адаптеров в Zope Component Architecture и в применении шаблона *Adapter* в классической книге ”Шаблоны проектирования”являются схожими. Но приемы использования адаптеров в ZCA обладают большей гибкостью и широтой применения по сравнению с классическим шаблоном. Шаблон *Adapter* задает лишь преобразование некоторого класса к какому-то другому интерфейсу, который требует сервис от экземпляров класса, с которыми он взаимодействует. Шаблон позволяет разным классам взаимодействовать друг с другом, даже если они до этого не могли это делать из-за несовместимости их интерфейсов. В подразделе *Мотивация* книги ”Шаблоны проектирования”авторы пишут: ”Часто адаптер отвечает за функциональность, которую не может предоставить адаптируемый класс”. Адаптеры ZCA фокусируются больше как раз на реализации новых функций, чем просто создание нового интерфейса для адаптируемого объекта. При этом адаптер в ZCA расширяет функциональность за счет новых методов. Будет, наверное, интересным заметить, что *Adapter* в ранних версиях дизайна проекта ZCA назывался *Feature*.[15]

Упомянутая в предыдущем абзаце цитата из книги Банды четырех заканчивается следующими словами: ”… которую не может предоставить адаптируемый класс”. Но в следующем предложении абзаца использовано словосочетание ”адаптируемого объекта”вместо ”адаптируемого класса так как авторы книги на самом деле пишут про два варианта адаптеров, вариант зависит от реализации. Первый вариант – это ”Адаптер класса а другой – ”Адаптер объекта”. Адаптер классов использует множественное наследование для

15

Дискуссия по переименованию *Feature* в *Adapter* в списке рассылки:

system=message
WARNING/2 in `izca-ru.txt`, line 1264
Definition list ends without a blank line; unexpected unindent. backrefs:

http://mail.zope.org/pipermail/zope3-dev/2001-December/000008.html

адаптации одного интерфейса к другому, с другой стороны, адаптер объекта опирается на композицию объектов. Адаптеры ZCA согласно книге Банды четырех следуют принципу адаптеру объекта, который использует делегирование (агрегирование) в качестве механизма композиции. Второй принцип объектно-ориентированного дизайна согласно Банде четырех выражается следующим образом: ”Предпочитайте объектную композицию наследованию классов”. Если хотите более детально познакомиться с приемами адаптации, прочтите книгу ”Шаблоны проектирования”.

Самой привлекательной стороной адаптеров ZCA – это явное использование интерфейсов, в виде специальных объектов Python, во время исполнения программы, а также реестра компонент. Адаптеры ZCA регистрируются в реестре компонент, через который другие объекты, затем, могут к ним получить доступ при помощи интерфейсов и механизмов поименования.

Глава 5. Утилиты

Вы уже знакомы с концепциями ”интерфейс, ”адаптер”и ”реестр компонент”. Иногда удобно регистрировать и отдельные объекты, которые ничего не адаптируют. Примерами таких объектов выступают соединения с базами данных, трансляторы XML, объекты, порождающие уникальные идентификаторы. Глобальные компоненты такого рода называются `утилитами`.

Утилиты – это объекты, которые обслуживают некоторый интерфейс, и к которым модули программ получают доступ по этим интерфейсам и регистрационным именам. Такой подход, с пользованием глобального реестра, позволяет регистрировать различные объекты и получать к ним доступ из различных частей программы, и при этом нет необходимости передавать эти объекты между модулями программы каким-либо специальным образом.

В программе нет необходимости регистрировать все глобальные объекты, смысл регистрации состоит в том, чтобы создать механизмы замены одних объектов, реализующих некоторый сервис, на другие, которые делают это как-то по-другому.

5.1. Простые утилиты

Утилиты могут быть зарегистрированы поименованными или нет. Пример поименованной утилиты рассмотрим в следующем разделе. Перед тем, как реализовать утилиту определим ее интерфейс. Следующий программный код определяет интерфейс *greeter* утилиты, которая будет приветствовать постояльцев:

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

class IGreeter(Interface):

    def greet(name):
        """Поприветствовать"""
```

Как и адаптеры, так и утилиты можно реализовать несколькими способами. Вот классическая реализация при помощи класса:

```
class Greeter(object):

    implements(IGreeter)

    def greet(self, name):
        return "Здравствуйте, " + name
```

В данном случае утилитой будет экземпляр класса *Greeter*. Чтобы утилитой можно было пользоваться, необходимо ее зарегистрировать. Для этого пользуется API ZCA. Утилита регистрируется при помощи метода реестра компонент `registerUtility`:

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

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

В этом примере утилита зарегистрирована как компонента, обслуживающая интерфейс *IGreeter*. Доступ к утилите получается по запросу обслуживаемого ей интерфейса, передав его функций *queryUtility* или *getUtility* в качестве первого параметра.

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

queryUtility(IGreeter).greet('Jack')
'Здравствуйте, Jack'

getUtility(IGreeter).greet('Jack')
'Здравствуйте,~Jack'
```

По аналогии с адаптерами, которые, как правило, являются классами, утилиты, как правило, являются экземплярами классов. Поэтому экземпляры утилит создаются единожды, а экземпляры адаптеров – всякий раз, как кто-то желает получить адаптер для объекта или нескольких объектов.

5.2. Поименованные утилиты

При регистрации компоненты-утилиты, как и адаптера, можно давать им имена. Утилита, зарегистрированная под каким-либо именем, называется поименованной утилитой.

Приведем пример регистрации поименованной утилиты *greeter*:

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

Экземпляр *greet* зарегистрирован как утилита, обслуживающая интерфейс *IGreeter*, под именем *new*. Теперь можно получить к ней доступ одной из функций *queryUtility* или *getUtility*:

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

    queryUtility(IGreeter, 'new').greet('Jill')
'Здравствуйте, Jill'

    getUtility(IGreeter, 'new').greet('Jill')
'Здравствуйте, Jill'
```

Заметим, что для получения доступа необходимо использовать имя *name* во втором аргументе функций.

Вызов функции *getUtility* без имени (второй аргумент) эквивалентно вызову этой функции с пустой строкой в качестве второго аргумента, так как это – значение по умолчанию. Если в данном примере реестру компонент задать поиск компоненты, чье имя – пустая строка, такая попытка поиска завершится неуспехом. Если компонента не найдена, то функция создают исключение `ComponentLookupError`. Она не будут вам возвращать какие-то другие компоненты, зарегистрированные под другими именами. Функции получения доступа к адаптерам *getAdapter* и *queryAdapter* функционируют аналогично.

5.3. Фабрики

Компоненты, называемые *фабриками* (`Factory`), – это утилиты, обслуживающие интерфейс `IFactory`.

Чтобы создать фабрику, сначала необходимо определить интерфейс объекта.

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

    class IDatabase(Interface):

        def getConnection():
            """Вернуть объект - соединение с базой данных"""
```

Вот пример заглушки, реализующей интерфейс *IDatabase*:

```
    class FakeDb(object):

        implements(IDatabase)

        def getConnection(self):
            return "connection"
```

Теперь можно создать фабрику при помощи `zope.component.factory.Factory`.

```
from zope.component.factory import Factory

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

Зарегистрируем фабрику.

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

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

Для получения доступа к фабрике, можно использовать следующий код:

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

Кроме того, можно и удобную использовать сокращенную версию запроса на порождение фабрикой конкретных объектов:

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

Глава 6. Дополнительные возможности адаптеров

В этом разделе рассмотрим дополнительные возможности, предоставляемые специальными видами адаптеров, в частности, *мультиадаптерами* (*multi adapter*), *адаптерами-подписчиками* (*subscription adapter*) и *обработчиками* (*handler*).

6.1. Мультиадаптеры

Простые адаптеры адаптируют только один объект, но, в принципе, компоненты могут адаптировать и несколько объектов. Если компонента адаптирует больше одного объекта, кто она называется *мультиадаптер* (*multi-adapter*).

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

class IAdapteeOne(Interface):
    pass

class IAdapteeTwo(Interface):
    pass

class IFunctionality(Interface):
    pass

class MyFunctionality(object):
    implements(IFunctionality)
    adapts(IAdapteeOne, IAdapteeTwo)

    def __init__(self, one, two):
        self.one = one
        self.two = two

from zope.component import getGlobalSiteManager
gsm = getGlobalSiteManager()

gsm.registerAdapter(MyFunctionality)

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

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

    one = One()
    two = Two()

    from zope.component import getMultiAdapter

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

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

6.2. Адаптеры-подписчики

В отличие от обычных адаптеров, адаптеры-подписчики, еще их называют просто *подписчиками*, используются, если есть необходимость реализовать возможность набора некоторого списка достаточно однотипных компонент-адаптеров, адаптирующих объект к одному и тому же интерфейсу. Полученный список адаптеров, затем, можно поэлементно перебрать и выполнить одну и ту же процедуру в каждом адаптере-подписчике. Результаты волнения процедур каким-либо образом объединяются и используются.

Рассмотрим задачу верификации объектов. Пусть задан набор объектов, и нам требуется оценить их соответствие некоторым требованиям. Определим интерфейс верификатора.

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

    class IValidate(Interface):

        def validate(ob):
            """Определить соответствие объекта стандарту

            Возвращает строку, характеризующую отклонение от
            стандарта. Пустая строка обозначает, что объект
            успешно прошел верификацию.
            """
```

Определим объекты-документы.

```
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
```

Теперь можно определить разные правила верификации документов. Например, можно потребовать, чтобы конспект-резюме документа был текстом из одной строки.

```
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 'Резюме должно состоять только из одной' +
            ' строки текста'
        else:
            return ''
```

Можно потребовать, чтобы основной текст документа состоял как минимум из 1000 символов.

```
class AdequateLength(object):

    adapts(IDocument)
    implements(IValidate)

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

```
        def validate(self):
            if len(self.doc.body) < 1000:
                return 'слишком короткий'
            else:
                return ''
```

Теперь надо зарегистрировать эти адаптеры-подписчики.

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

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

И теперь зарегистрированные подписчики можно использовать для верификации объектов.

```
from zope.component import subscribers

doc = Document("A text\nDocument", "lorum")
[adapter.validate()
 for adapter in subscribers([doc], IValidate)
 if adapter.validate()]
['Резюме должно состоять только из одной строки текста', 'слишком
↪ короткий']

doc = Document("A text\nDocument", "ipsum" * 1000)
[adapter.validate()
 for adapter in subscribers([doc], IValidate)
 if adapter.validate()]
['Резюме должно состоять только из одной строки текста']

doc = Document("A Document", "dolor")
[adapter.validate()
 for adapter in subscribers([doc], IValidate)
 if adapter.validate()]
['слишком короткий']
```

6.3. Обработчики

Обработчики – это фабрики адаптеров-подписчиков, которые ничего не возвращают. Они просто выполняют некоторую процедуру, как только им передается управление. Основной вид использования обработчиков – это реализация реакции на возникающие в приложении события. Обработчики по

своей сути являются подписчиками на события, и их иногда называют *адаптеры, подписанные на события*.

Подписчики на события отличаются от других адаптеров-подписчиков тем, что объект, вызывающий обработчика, не ожидает от него никакого прямого взаимодействия. Например, объект, инициирующий событие не ожидает от обработчиков этого события каких-либо значений в ответ. Подписчикам нет необходимости обслуживать какой-либо API для вызвавших их компонентов, поэтому более естественно реализовывать подписчики при помощи функций, а не классов. В системе управления контентом, например, можно автоматизировать добавление времени создания документа в виде поля этого документа.

```
import datetime

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

В предыдущем примере определена функция, которая получает событие (*event*) и выполняет процедуру добавления данных в соответствующий объект. Результат, как видите, не возвращается. Обработчики – это частные случаи подписчиков, которые адаптируют события к ”ничто”. Вся работа выполняется, когда этот адаптер-”фабрика”был вызван. Для регистрации обработчиков в ZCA APIs используются специальные функции.

Чтобы продемонстрировать регистрацию и использование вышеупомянутого обработчика, сначала создадим событие *документ-создан* (*document-created*).

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

class IDocumentCreated(Interface):

    doc = Attribute(”Документ, который был создан”)

class DocumentCreated(object):

    implements(IDocumentCreated)

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

Также немного дополним текст нашего обработчика.

```
    from zope.component import adapter

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

Добавленная конструкция помечает обработчик интерфейсом *IDocumentCreated* компоненты-события *DocumentCreated*.

Теперь необходимо обработчик зарегистрировать.

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

    gsm.registerHandler(documentCreated)
```

Теперь можно создать компоненту-событие и инициировать его обработку функцией *handle*, которая вызовет все зарегистрированные обработчики для данного события.

```
    from zope.component import handle

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

Глава 7. Как ZCA используется в Zope

Компонентная архитектура Zope используется как в Zope 3 и более новых версиях, так и в Zope 2. В этом разделе предложен обзор применения ZCA в Zope.

7.1. ZCML

Язык **Zope Configuration Markup Language (ZCML)** (Язык разметки для конфигурирования Zope) – это язык задания конфигурации программной системе, базирующийся на стандарте XML. Вместо того, чтобы использовать API Python для регистрации компонент, можно использовать язык ZCML. Но для этого сначала надо установить несколько пакетов, от которых зависит функционирование языка.

Необходимые зависимости догружаются во время установки пакета `zope.componer` в варианте с поддержкой ZCML. Загрузка пакетов выполняется утилитой командной строки easy_install следующим образом:

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

Каждый файл конфигурации и регистрации компонент ZCML должен начинаться с директивы (тега) `configure` с соответствующей декларацией пространства имен.

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

</configure>
```

Директива *adapter* предназначена для регистрации адаптеров.

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

Атрибуты *provides* и *for* являются необязательными, если необходимая информация ранее была представлена в реализации.

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

Для того, чтобы зарегистрировать компоненту как поименованный адаптер надо использовать атрибут *name*.

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

Утилиты регистрируются директивой *utility*. Для этого используется следующая конструкция:

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

Атрибут *provides* не является обязательным, если необходимая информация представлена в реализации.

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

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

Поименованные утилиты регистрируются также, только необходимо использовать атрибут *name*.

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

Вместо регистрации утилиты как объекта (экземпляра) можно зарегистрировать его фабрику.

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

7.2. Замещения

При регистрации компонент методами `register*` API Python реализуется следующее правило: компонента, которая была зарегистрирована последней заменяет (замещает) собой ранее зарегистрированную компоненту, если они обе зарегистрированы с одним и тем же набором аргументов функций-регистраторов. Рассмотрим следующий пример:

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

    class IA(Interface):
        pass

    class IP(Interface):
        pass

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

    from zope.component import getGlobalSiteManager
    gsm = getGlobalSiteManager()

    class AP(object):

        implements(IP)
        adapts(IA)

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

    class AP2(object):

        implements(IP)
        adapts(IA)

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

    class A(object):

        implements(IA)

    a = A()
    ap = AP(a)

    gsm.registerAdapter(AP)

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

Теперь, если зарегистрировать еще один адаптер, то уже существующий будет замещен.

```
    gsm.registerAdapter(AP2)

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

Если же регистрировать компоненты при помощи ZCML, то вторая регистрация создаст конфликт, что проявится в виде исключения. Этот принцип позволяет контролировать ситуации, когда нужный зарегистрированный адаптер по ошибке замещается каким-то другим. В результате удается избегать необходимость отслеживать ошибки в работе проектируемой программной системы. Получается, что использование ZCML способствует даже упрощению разработки.

Иногда, все-таки, необходимо заменять существующие регистрации. Для этого в ZCML есть директива `includeOverrides`. Используя эту директиву, можно указывать, что в отдельном файле (атрибут *file*) находятся регистрации, замещающие собой имеющиеся:

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

7.3. Адаптер NameChooser

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

Адаптер NameChooser – это адаптер, который подбирает уникальное имя для некоторого объекта внутри контейнера.

Регистрация такого адаптера выглядит следующим образом:

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

Из данной конструкции видно, что адаптируемый объект обеспечивает `IWriteContainer`, а адаптер – `INameChooser`.

Адаптер *NameChooser* очень удобен программистам Zope. Реализация `IWriteContainer` в Zope 3 распределена, в основном, между классами пакетов `zope.app.container.BTreeContainer` и `zope.app.folder.Folder`. Обычно в своих программах надо наследовать от этих классов их функции для создания собственных классов-контейнеров. Если бы разработчики ранее не реализовали интерфейс `INameChooser` и адаптер `NameChooser`, тогда вам пришлось бы вновь создавать их для каждой реализации отдельно.

7.4. Адаптер LocationPhysicallyLocatable

Пакет: `zope.location.traversing.LocationPhysicallyLocatable`

Этот адаптер часто используется при программировании приложений в среде Zope 3, но вызывается обычно при помощи специального API в пакете `zope.traversing.api`. (Редкий старый код до сих пор использует функции `zope.app.zapi`, которые добавляют еще один слой косвенных вызовов для этого адаптера)

Регистрация данного адаптера выполняется следующим образом:

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

Интерфейс, который обеспечивает компонента и адаптируемый интерфейс задаются в реализации, начало которой выглядит вот так:

```
class LocationPhysicallyLocatable(object):
    """Обеспечить информацию о местоположении объектов
    """
    zope.component.adapts(ILocation)
    zope.interface.implements(IPhysicallyLocatable)
    ...
```

Как правило, все объекты, хранимые в базах данных, в приложении Zope 3 обеспечивают интерфейс `ILocation`. У этого интерфейса есть только два атрибута, `__parent__` и `__name__`. Атрибут `__parent__` соответствует родительскому узлу в иерархии адресов, а `__name__` – это название объекта, которое уникально среди множества объектов с одним родительским узлом.

Интерфейс `IPhysicallyLocatable` содержит четыре метода: `getRoot`, `getPath`, `getName`, и `getNearestSite`.

- `getRoot` – функция, которая возвращает объект, соответствующий корневому узлу.
- `getPath` возвращает физический путь к объекту в виде строки.
- `getName` возвращает последний сегмент физического пути.
- `getNearestSite` возвращает объект-сайт, в котором содержится адаптируемый объект, как правило это регистр компонент, привязанный к конкретному сайту (интернет-приложению, реализованному на одном сервере Zope 3); если он сам является сайтом, то он сам и возвращается.

Если вы решите заняться изучением среды Zope 3, вы увидите, что компоненты, о которых идет сейчас речь, являются важными сущностями, и они используются очень и очень часто. Для полного осознания всей красоты такого подхода необходимо также разобраться, как в системе Zope 2 осуществляется поиск корневого объекта и как это реализовано: почти в каждом объекте-контейнере есть метод `getPhysicalRoot`.

7.5. Адаптер DefaultSized

Пакет: `zope.size.DefaultSized`

Этот адаптер – просто реализация интерфейса `ISized` по умолчанию. Адаптер ”знает”как адаптировать все виды объектов, т. е. зарегистрирован как адаптер всех объектов. Если вам потребуется зарегистрировать этот адаптер для какого-либо интерфейса, то придется уже существующую регистрацию замещать новой.

Регистрация этого адаптера в Zope сделана следующим образом:

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

Здесь видно, что адаптируемый интерфейс задан как *, т. е. *DefaultSized* адаптирует любые разновидности объектов.

Интерфейс `ISized` очень прост и включает спецификации всего двух методов.

```
class ISized(Interface):

    def sizeForSorting():
        """Возвращает двойку (basic_unit, amount).

        Используется для сортировки разнородных объектов,
        характеризующихся некоторым размером. Значение 'amount'
        требуется только для сортировки объектов с одной и той же
        единицей измерения 'basic_unit'."""

    def sizeForDisplay():
        """Возвращает строку, представляющую размер объекта.
        """
```

Пример специально зарегистрированного адаптера `ISized` для интерфейса `IZPTPage` можно посмотреть в пакете `zope.app.zptpage`.

7.6. Утилита ZopeVersionUtility

Пакет: `zope.app.applicationcontrol.ZopeVersionUtility`
Утилита сообщает версию выполняющегося процесса Zope.
Регистрация выполняется следующим образом:

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

Интерфейс, который обслуживает эта утилита, называется `IZopeVersion`, он специфицирует только один метод, называемый `getZopeVersion`. Этот метод возвращает строку, характеризующую версию программного обеспечения среды Zope, включая всевозможную информацию, например, версию исходного кода в репозитарии Subversion.

Реализация утилиты по умолчанию, `ZopeVersionUtility`, получает информацию о версии Zope из файла `version.txt` в директории *zope/app*. Если система Zope была запущена в слепке (checkout) сервера Subversion, то утилита покажет последний номер ревизии. Если ни того ни другого источника информации не удалось найти, утилита вернет строку *Development/Unknown*.

7.7. Адаптеры–«заместители»

FIXME: Вероятно не в этом разделе это должно быть.

Практически в любом приложении предполагается наличие системы пользователей и ролей, где каждой роли назначается набор функций, допустимых к исполнению. Например, руководитель предприятия получает доступ к результатам анализа финансового состояния предприятия. Специалист-аналитик проводит этот анализ на основе данных базы данных предприятия, а работники имеют право вводить данные в базу данных согласно структуру предприятия. При этом множества функций, выполняемых пользователями (ролями), как правило, не пересекаются. Такое разделение позволяет уменьшить количество ошибок в базе данных и структуре документооборота на предприятии.

В рамках компонентной модели система ролей реализуется при помощи адаптеров…

7.7.1. Распределенные приложения

Иногда бывает так, что ресурсов одной рабочей станции достаточно для реализации всех необходимых функций программной системы…. Для этого необходимо выделять отдельные процессы и переносить их на другие

рабочие станции. Типичным примером являются сервера баз данных. Серьезный сервер баз данных требует определенных характеристик аппаратного обеспечения сервера, структуры файловой системы, свойств операционной системы и т. п.

Другим примером, когда необходимо выделять функции в отдельные процессы, выступает процесс интеграции разрабатываемой программной системы со сторонними приложениями. …

Здесь мы рассмотрим пример выделения компоненты в отдельный процесс…

7.7.2. Протокол и библиотеки XML-RPC

RPC stands for Remote Procedure Call. As its name indicates, it is a mechanism to call a procedure or a function available on a remote computer. RPC is a much older technology than the Web. Effectively, RPC gives developers a mechanism for defining interfaces that can be called over a network. These interfaces can be as simple as a single function call or as complex as a large API. What is XML-RPC ?

XML-RPC is among the simplest and most foolproof web service approaches that makes it easy for computers to call procedures on other computers.

XML-RPC permits programs to make function or procedure calls across a network.

XML-RPC uses the HTTP protocol to pass information from a client computer to a server computer.

XML-RPC uses a small XML vocabulary to describe the nature of requests and responses.

XML-RPC client specifies a procedure name and parameters in the XML request, and the server returns either a fault or a response in the XML response.

XML-RPC parameters are a simple list of types and content - structs and arrays are the most complex types available.

XML-RPC has no notion of objects and no mechanism for including information that uses other XML vocabulary.

With XML-RPC and web services, however, the Web becomes a collection of procedural connections where computers exchange information along tightly bound paths.

XML-RPC emerged in early 1998; it was published by UserLand Software and initially implemented in their Frontier product.

Why XML-RPC ?

If you need to integrate multiple computing environments, but don’t need to share complex data structures directly, you will find that XML-RPC lets you establish communications quickly and easily.

Even if you work within a single environment, you may find that the RPC approach makes it easy to connect programs that have different data models or processing expectations and that it can provide easy access to reusable logic.

XML-RPC is an excellent tool for establishing a wide variety of connections between computers.

XML-RPC offers integrators an opportunity to use a standard vocabulary and approach for exchanging information.

XML-RPC's most obvious field of application is connecting different kinds of environments, allowing Java to talk with Perl, Python, ASP, and so on.

XML-RPC Technical Overview

XML-RPC consists of three relatively small parts:

XML-RPC data model : A set of types for use in passing parameters, return values, and faults (error messages).

XML-RPC request structures : An HTTP POST request containing method and parameter information.

XML-RPC response structures : An HTTP response that contains return values or fault information.

We will study all these three components in the next three chapters.

7.7.3. Реализация распределенного приложения

Глава 8. Интерфейсы пользователя

Как и любую другую программная компонента интерфейс пользователя тоже имеет смысл разрабатывать как компоненту приложения. Естественно здесь есть свои особенности.

8.1. Шаблон проектирования Model-View-Controller

The Model-View-Control (MVC) pattern, originally formulated in the late 1970s, is a software architecture pattern built on the basis of keeping the presentation of data separate from the methods that interact with the data. In theory, a well-developed MVC system should allow a front-end developer and a back-end developer to work on the same system without interfering, sharing, or editing files either party is working on.

Even though MVC was originally designed for personal computing, it has been adapted and is widely used by web developers due to its emphasis on separation of concerns, and thus indirectly, reusable code. The pattern encourages the development of modular systems, allowing developers to quickly update, add, or even remove functionality.

In this article, I will go the basic principles of MVC, a run through the definition of the pattern and a quick example of MVC in PHP. This is definitely a read for anyone who has never coding with MVC before or those wanting to brush up on previous MVC development skills. Understanding MVC

The pattern's title is a collation of its three core parts: Model, View, and Controller. A visual representation of a complete and correct MVC pattern looks like the following diagram:

MVC Process

The image shows the single flow layout of data, how it's passed between each component, and finally how the relationship between each component works.
Model

The Model is the name given to the permanent storage of the data used in the overall design. It must allow access for the data to be viewed, or collected and written to, and is the bridge between the View component and the Controller component in the overall pattern.

One important aspect of the Model is that it's technically "blind" – by this I mean the model has no connection or knowledge of what happens to the data when it is passed to the View or Controller components. It neither calls nor seeks a response from the other parts; its sole purpose is to process data into its permanent storage or seek and prepare data to be passed along to the other parts.

The Model, however, cannot simply be summed up as a database, or a gateway to another system which handles the data process. The Model must act as a gatekeeper

to the data itself, asking no questions but accepting all requests which comes its way. Often the most complex part of the MVC system, the Model component is also the pinnacle of the whole system since without it there isn't a connection between the Controller and the View. View

The View is where data, requested from the Model, is viewed and its final output is determined. Traditionally in web apps built using MVC, the View is the part of the system where the HTML is generated and displayed. The View also ignites reactions from the user, who then goes on to interact with the Controller. The basic example of this is a button generated by a View, which a user clicks and triggers an action in the Controller.

There are some misconceptions held about View components, particularly by web developers using the MVC pattern to build their application. For example, many mistake the View as having no connection whatsoever to the Model and that all of the data displayed by the View is passed from the Controller. In reality, this flow disregards the theory behind the MVC pattern completely. Fabio Cevasco's article The CakePHP Framework: Your First Bite demonstrates this confused approach to MVC in the CakePHP framework, an example of the many non-traditional MVC PHP frameworks available:

"It is important to note that in order to correctly apply the MVC architecture, there must be no interaction between models and views: all the logic is handled by controllers"

Furthermore, the description of Views as a template file is inaccurate. However, as Tom Butler points out, this is not one person's fault but a multitude of errors by a multitude of developers which result in developers learning MVC incorrectly. They then go on to educate others incorrectly. The View is really much more than just a template, however modern MVC inspired frameworks have bastardised the view almost to the point that no one really cares whether or not a framework actually adheres to the correct MVC pattern or not.

It's also important to remember that the View part is never given data by the Controller. As I mentioned when discussing the Model, there is no direct relationship between the View and the Controller without the Model in between them. Controller

The final component of the triad is the Controller. Its job is to handle data that the user inputs or submits, and update the Model accordingly. The Controller's life blood is the user; without user interactions, the Controller has no purpose. It is the only part of the pattern the user should be interacting with.

The Controller can be summed up simply as a collector of information, which then passes it on to the Model to be organized for storage, and does not contain any logic other than that needed to collect the input. The Controller is also only connected to a single View and to a single Model, making it a one way data flow system, with handshakes and signoffs at each point of data exchange.

It's important to remember the Controller is only given tasks to perform

when the user interacts with the View first, and that each Controller function is a trigger, set off by the user's interaction with the View. The most common mistake made by developers is confusing the Controller for a gateway, and ultimately assigning it functions and responsibilities that the View should have (this is normally a result of the same developer confusing the View component simply as a template). Additionally, it's a common mistake to give the Controller functions that give it the sole responsibility of crunching, passing, and processing data from the Model to the View, whereas in the MVC pattern this relationship should be kept between the Model and the View. MVC in PHP

It is possible to write a web application in PHP whose architecture is based on the MVC pattern. Let's start with a bare bones example:

We have our project started with some very basic classes for each part of the pattern. Now we need to set up the relationships between them:

As you can see in the example above, we don't have any Controller-specific functionality because we don't have any user interactions defined with our application. The View holds all of the functionality as the example is purely for display purposes.

Let's now expand the example to show how we would add functionality to the controller, thereby adding interactivity to the application:

We've enhanced the application with some basic functionality. Setting up the relationship between our components now looks like this:

Run the code and when you click on the link you'll be able to see the string change its data. Conclusion

We've covered the basic theory behind the MVC pattern and have produced a very basic MVC application, but we still have a long way to go before we get into any nitty-gritty functionality.

Next up in the series we'll cover some of the choices you face when trying to create a true MVC application on the web in PHP. Stay tuned!

8.2. Библиотека GTK+

GTK+ is a widget toolkit. Each user interface created by GTK+ consists of widgets. This is implemented in C using GObject, an object-oriented framework for C. Widgets are organized in a hierachy. The window widget is the main container. The user interface is then built by adding buttons, drop-down menus, input fields, and other widgets to the window. If you are creating complex user interfaces it is recommended to use GtkBuilder and its GTK-specific markup description language, instead of assembling the interface manually. You can also use a visual user interface editor, like Glade.

GTK+ is event-driven. The toolkit listens for events such as a click on a button, and passes the event to your application.

This chapter contains some tutorial information to get you started with GTK+ programming. It assumes that you have GTK+, its dependencies and a C compiler installed and ready to use. If you need to build GTK+ itself first, refer to the Compiling the GTK+ libraries section in this reference.

In a GTK+ application, the purpose of the main() function is to create a GtkApplication object and run it. In this example a GtkApplication pointer named app is called and then initialized using gtk_application_new().

When creating a GtkApplication you need to pick an application identifier (a name) and input to gtk_application_new() as parameter. For this example org.gtk.example is used but for choosing an identifier for your application see this guide. Lastly gtk_application_new() takes a GApplicationFlags as input for your application, if your application would have special needs.

Next the activate signal is connected to the activate() function above the main() functions. The activate signal will be sent when your application is launched with g_application_run() on the line below. The gtk_application_run() also takes as arguments the pointers to the command line arguments counter and string array; this allows GTK+ to parse specific command line arguments that control the behavior of GTK+ itself. The parsed arguments will be removed from the array, leaving the unrecognized ones for your application to parse.

Within g_application_run the activate() signal is sent and we then proceed into the activate() function of the application. Inside the activate() function we want to construct our GTK window, so that a window is shown when the application is launched. The call to gtk_application_window_new() will create a new GtkWindow and store it inside the window pointer. The window will have a frame, a title bar, and window controls depending on the platform.

A window title is set using gtk_window_set_title(). This function takes a GtkWindow* pointer and a string as input. As our window pointer is a GtkWidget pointer, we need to cast it to GtkWindow*. But instead of casting window via (GtkWindow*), window can be cast using the macro GTK_WINDOW(). GTK_WINDOW() will check if the pointer is an instance of the GtkWindow class, before casting, and emit a warning if the check fails. More information about this convention can be found here.

Finally the window size is set using gtk_window_set_default_size and the window is then shown by GTK via gtk_widget_show_all().

When you exit the window, by for example pressing the X, the g_application_run() in the main loop returns with a number which is saved inside an integer named “status”. Afterwards, the GtkApplication object is freed from memory with g_object_unref(). Finally the status integer is returned and the GTK application exits.

While the program is running, GTK+ is receiving events. These are typically input events caused by the user interacting with your program, but also things like messages from the window manager or other applications. GTK+ processes these

and as a result, signals may be emitted on your widgets. Connecting handlers for these signals is how you normally make your program do something in response to user input.

The following example is slightly more complex, and tries to showcase some of the capabilities of GTK+.

In the long tradition of programming languages and libraries, it is called Hello, World.

As seen above, example-1.c builds further upon example-0.c by adding a button to our window, with the label “Hello World”. Two new GtkWidget pointers are declared to accomplish this, button and button_box. The button_box variable is created to store a GtkButtonBox which is GTK+’s way of controlling the size and layout of buttons. The GtkButtonBox is created and assigned to gtk_button_box_new() which takes a GtkOrientation enum as parameter. The buttons which this box will contain can either be stored horizontally or vertically but this does not matter in this particular case as we are dealing with only one button. After initializing button_box with horizontal orientation, the code adds the button_box widget to the window widget using gtk_container_add().

Next the button variable is initialized in similar manner. gtk_button_new_with_label() is called which returns a GtkButton to be stored inside button. Afterwards button is added to our button_box. Using g_signal_connect the button is connected to a function in our app called print_hello(), so that when the button is clicked, GTK will call this function. As the print_hello() function does not use any data as input, NULL is passed to it. print_hello() calls g_print() with the string “Hello World” which will print Hello World in a terminal if the GTK application was started from one.

After connecting print_hello(), another signal is connected to the “clicked” state of the button using g_signal_connect_swapped(). This functions is similar to a g_signal_connect() with the difference lying in how the callback function is treated. g_signal_connect_swapped() allow you to specify what the callback function should take as parameter by letting you pass it as data. In this case the function being called back is gtk_widget_destroy() and the window pointer is passed to it. This has the effect that when the button is clicked, the whole GTK window is destroyed. In contrast if a normal g_signal_connect() were used to connect the “clicked” signal with gtk_widget_destroy(), then the button itself would have been destroyed, not the window. More information about creating buttons can be found here.

The rest of the code in example-1.c is identical to example-0.c. Next section will elaborate further on how to add several GtkWidgets to your GTK application.

8.2.1. Дизайнер интерфесов Glade

Мы не будем далее вдаваться в подробности функционирования элементов управления библиотеки, а воспользуемся дизайнером интерфейсов Glade (лезвие).

Glade is a RAD tool to enable quick and easy development of user interfaces for the GTK+ toolkit and the GNOME desktop environment.

The user interfaces designed in Glade are saved as XML, and by using the GtkBuilder GTK+ object these can be loaded by applications dynamically as needed.

By using GtkBuilder, Glade XML files can be used in numerous programming languages including C, C++, C#, Vala, Java, Perl, Python,and others.

OK, since it looks like you gave me a chance to show how I usually work, let's start.

Today's work will be separated into two sections:

Creating "blueprint" of our application's GUI based on requirements. Writing down widget tree, based on blueprint from previous section.

Now, what are requirements for our application. Obviously, it needs to be able to display data in numeric and graphic form. We also want to be able to add new, delete, rearrange and modify points. We also want to be able to display point markers, connecting lines or both on the chart. And this is about it for initial requirements. We'll leave some space for future expansion in our plans, just in case if we decide to add anything to it later.

Now comes the fun part - drawing interface. I prefer to do this on paper using pencil, but feel free to experiment. For sample project, I came up with this design:

Mock-up of interface

Dark gray rectangles represent scroll bars. What do you think? Do I suck as interface designer or do I really suck as interface designer?;)

Now for the last thing we need to do today: writing down widget tree. In GTK+, everything starts with top-level GtkWindow, which will serve as a root of our widget tree.

Our main window will be split into four vertical sections: one for menu bar, one for tool bar, one for central part where all the action will happen and last one for status bar. Since our main window (GtkWindow) can hold only one child widget, we need GtkVBox in which we will pack, from bottom to top: GtkStatusbar, widget for central part, GtkToolbar and GtkMenuBar. Why did I say that we'll be packing from bottom? I'll explain this in next post, when we'll be playing with Glade3.

Central part will need to be further divided into horizontal sections: one for data table, one for point controls and one for display area. So this will require GtkHBox. What widgets will we need inside it? For data table we'll use GtkTreeView, which is packed inside GtkScrolledWindow to enable scrolling. For point controls

we'll need GtkVButtonBox that will house our buttons.

Now for the display area. We again have two parts: upper part that will actually display chart; and bottom part that will hold check buttons. So we'll need another GtkVBox to hold the whole section. Bottom part will be represented by GtkHButtonBox with GtkToggleButtons, while upper part deserves another paragraph;).

Why another paragraph? Because we'll need to add quite a few widgets in order to get desired layout. As you can see on my drawing, I want to have graph area centered in display part. In order to be able to add zoom to chart area, we also need some kind of scrolling widget. And how to assemble all this together? First we'll add GtkScrolledWindow to GtkVBox from previous paragraph. To make contents scrollable, we need to pack GtkViewport inside GtkScrolledWindow. Inside GtkViewport we'll add GtkAlignment, which will take care of centering the chart area. Inside GtkAlignment we'll add GtkFrame, which will add a shadow to chart area. Finally, we add GtkDrawing area inside GtkFrame. And we're done.

There is just one thing I would like to explain today. When adding GtkTreeView to GtkScrolledWindow I haven't used GtkViewport as an adapter widget, while adding GtkAlignment did require one. Why? When it comes to GtkScrolledWindow, there are two kinds of widgets: the ones that support scrolling natively and the ones that don't. GtkTreeView, GtkTextView, GtkIconView and GtkViewport do support scrolling and can be added into GtkScrolledWindow directly. All other widgets need GtkViewport as an adapter.

I hope this starting lesson wasn't too tough and you'll join me next time when we'll fire up Glade3 and do some property mangling.

8.2.2. Конструирование интерфейса

In this part of tutorial, we'll create application's GUI using Glade3 according to our blueprint and widget tree. To check how our GUI looks like, we'll also write a minimalistic application.

I'll be using Glade3-3.6.7 in this tutorial, but any version from 3.6 series will do. Our project will be saved as a GtkBuilder project and we'll use GtkBuilder to create our interface at runtime. Glade3-3.4.5 cannot be used when following this tutorial, since we'll use some new features of Glade3 that were introduced in 3.6 series.

Contents

Glade3 tutorial (1) - Introduction Glade3 tutorial (2) - Constructing interface Glade3 tutorial (3) - Size negotiation Glade3 tutorial (4) - GtkTreeView data backend Glade3 tutorial (5) - Modifying widget tree Glade3 tutorial (6) - Signals

Constructing interface

I tried to document each step in GUI creation with screenshot and instructions. Video of the actual process can be found on ScreenToaster site.

When you start Glade3 without opening project, you're greeted by two windows: main application window that we'll be using to create interface and preferences dialog that lets you set some project settings.

You can see from this screen that we'll be using GtkBuilder project format with object names being unique across the whole project. We won't be using any images in this project, so resource location is not important to us. Lastly, this project will be compatible with GTK+ >= 2.14, which should make it usable on most distributions out there. You can check your project for any incompatibilities from this dialog too, but since we're starting new poject, this is not needed. Last thing is to click Close and we're done with initial setup.

Now it's time to place initial toplevel window into project. Just click "Window" icon in "Toplevels" category and you should see something like this:

Now we'll set window title to "Charter" and default size to 600 x 400 px.

Now we need to switch to "Common" tab in properties section and set "Border width" to 6 px.

Last thing we need to do with main window is to connect gtk_main_quit function to "destroy" signal. This will close our application when we'll click on close button. I'll talk more about signals in one of the following posts, so no more details will be given here.

With our main window finished, we need to add GtkVBox to it. In previous post, we planed to add four widgets inside this box, but since menu bar and tool bar will be created using GtkUIManager and manually inserted from code, we only need two. So when we're asked about number of elements, we enter 2.

Now we'll add status bar to application. Click status bar icon and insert it into bottom of the box. Your GUI should look something like this:

Now we need to make sure that status bar is packed from bottom to top. Why is this important? If we would to pack status bar from top to bottom, it would be impossible to add menu and tool bar at the top of the application. How to ensure that status bar is packed from bottom to top? Select "Packing tab" and select "End" as pack type and set "Position" to 0. What does this do? It instructs Glade to pack status bar as first element from bottom to top.

Now run this code and be amazed;). Not too bad, but nothing special either. Try resizing window to see how compartments behave when resized. Do you like it? I don't either, so join me next time when we'll be dealing with space alocation/requisition in detail.

And we're done. If you need more detailed process of clicking/changing properties, head to ScreenToaster site, where you can watch video of the whole thing. Hope you'll find it useful.

Finished glade file can be obtained from here: tut.glade.

8.3. Разработка контроллера

Контроллер по своей сути является мультиадаптером компонент модели (Model) и интерфейса (View), причем адаптером, который ничего не делает, кроме обеспечения взаимодействия модели и интерфейса. То есть контроллер – это обработчик (handler) и не более того. Запрограммируем его в соответствии с этим.

Глава 9. Пример приложения

> **Note**
> Эта часть не еще не завершена. Автор принимает предложения по ее развитию !

В этом разделе демонстрируется проектирование оконного приложения, используя библиотеку PyGTK для реализации интерфейса пользователя совместно с ZCA. Приложение использует два различных механизма постоянного хранения объектов. Первый – это объектно-ориентированная база данных ZODB, а вторая – реляционная база данных SQLite. Однако, при установке конкретного приложения только одна база данных будет использоваться. Использование двух механизмов постоянного хранения объектов позволяет продемонстрировать как ZCA собирает компоненты и связывает их в единое приложение. В сходном коде приложения преобладает реализация интерфейса пользователя, основывающаяся не библиотеке PyGTK.

При разработке больших приложений удобно использовать ZCA, так как он обеспечивает и управляемость процессом разработки комплексных приложений, и возможность расширения набора функций программной системы, встраивания и замену (pluggablity) компонент. Непосредственное использование объектов Python, как правило, не создает подобных условий.

Вообще ZCA может с одинаковым успехом использоваться как для разработки ВЕБ-, так и для оконных и других приложений. В каждом конкретно случае необходимо определиться с принципом организации регистрации компонент: в каком модуле какие компоненты будут регистрироваться. В рассматриваемом примере функции регистрации компонент задаются в отдельных модулях, и, затем, через импорт этих модулей производится их регистрация. Главный регистрирующий модуль компоненты – *register.py*.

Исходный код готового приложения находится по следующему адресу: `http://www.muthukadan.net/downloads/zcalib.tar.bz2`

9.1. Функции приложения

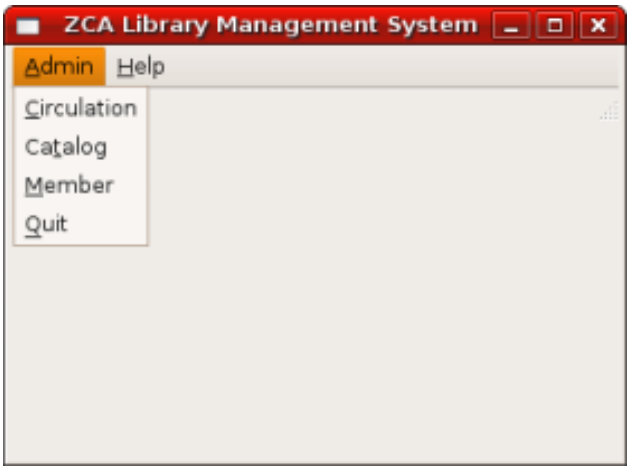
Приложение, о котором идет речь – это система управления библиотекой с некоторым минимальным набором функций. Основные требования к приложению следующие:

- Добавление читателей, идентифицируя их уникальным номером и именем.
- Добавление книг, помечая их баркодом, и указывая автора и название.

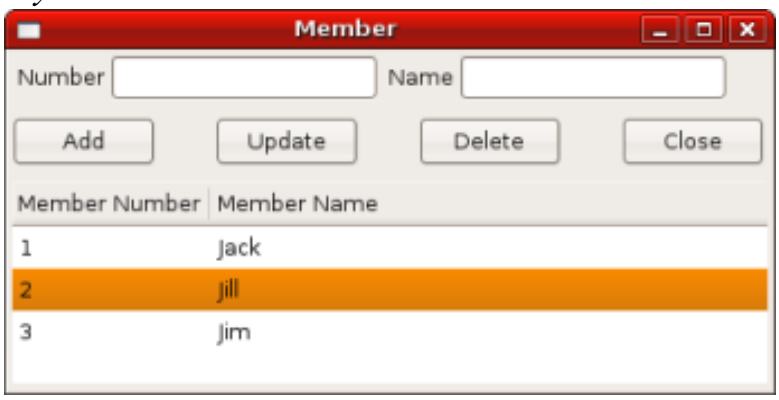
- Выдача книг на руки читателям.

- Учет книг, возвращенных читателями.

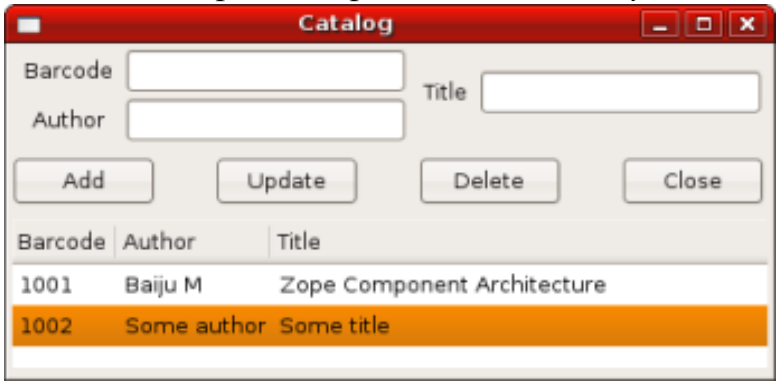
Дизайн интерфейса пользователя сделаны таким, чтобы основные функции были доступны из одного окна. Основное окно имеет следующий вид:



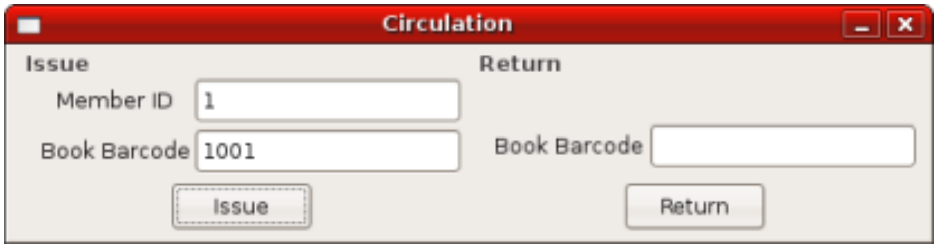
В приложении создано специальное окно для управления читателями. При помощи этого окна реализуются функции *добавление*, *обновление* данных читателя и *удаление* читателя:



Аналогично предыдущему окну, интерфейс управления базой данных книг позволяет *добавлять*, *редактировать* данные и *удалять* книги:



Окно управления книжным фондом реализует функции *выдачи* и *возврата* книг читателями:



9.2. Обзор кода PyGTK

Почти все приложение – это программный код, использующий PyGTK. Структура программ реализации всех окон схожа, общий вид окон и приложения разработан при помощи дизайнера оконных интерфейсов Glade GUI builder. Чтобы использовать разработанный интерфейс в программном коде необходимо основным элементам управления поименовать. В главном меню все пункты поименованы следующим образом: ”circulation ”catalog ”member ”quit”и ”about”.

Класс `gtk.glade.XML` используется для трансляции и интерпретации файлов glade, в результате его деятельности создаются элементы управления интерфейса пользователя. Вот пример трансляции и получения доступа к элементу управления:

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

В модуле mainwindow.py находится код, аналогичный следующему:

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

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

Название элемента управления, соответствующего главному окну, – *mainwindow*. Аналогичным образом можно получать доступ и к другим элементам управления.

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

Затем с элементами управления связываются обработчики возникающих событий.

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

Событие *delete_event* возникает, когда пользователь пытается закрыть окно при помощи соответствующей кнопки. Событие *activate* инициируется, когда пользователь выбрал какой-либо пункт меню. В качестве обработчиков событий выступают функции, вызываемые в момент возникновения события.

В приведенном фрагменте кода видно, что событие *delete_event* главного окна соединено с методом *on_delete_event*. Событие *activate* элемента *quit* также соединено с этим же методом.

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

Функция *on_delete_event* просто вызывает функцию *main_quit*.

9.3. Исходный код приложения

Модуль *zcalib.py*:

```
import registry
import mainwindow

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

В тексте импортируются модули *registry* и *mainwindow*. Затем модуль *registry* инициализируется и вызывается функция *main* модуля *mainwindow*. Если пользователь попытается завершить программу при помощи комбинации клавиш *Ctrl+C*, программа закончит свое исполнение. Чтобы заблокировать такое поведение программы, перехватим исключение *KeyboardInterrupt*.

Модуль *registry.py*:

```
import sys
from zope.component import getGlobalSiteManager

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


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

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

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

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

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

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

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

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

```
                        IDbOperation)

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

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

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

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

Обратите внимание на функцию *initialize*, которая вызывается в главном модуле *zcalib.py*. Функция сначала узнает, какую базу данных надо использовать: реляционную (RDB) или объектную (ODB). Указание варианта базы данных осуществляется при помощи функции *check_use_relational_db*. Если в командной строке задать опцию *-r*, будет запущена *initialize_rdb*, иначе – *initialize_odb*. Если выбрана RDB, то будут установлены компоненты, реализующие механизм RDB. В обратном случае – компоненты для ODB.

Модуль *mainwindow.py*:

```
import os
import gtk
import gtk.glade

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

class MainWindow(object):
```

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

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

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

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

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

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

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

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

    def on_about_activate(self, *args):
        pass

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

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

Функция *main* создает экземпляр класса *MainWindow*, который инициализирует все элементы управления.

Модуль *memberwindow.py*:

```
import os
import gtk
import gtk.glade

from zope.component import getAdapter

from components import Member
from interfaces import IDbOperation


class MemberWindow(object):

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

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

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

        self.initialize_list()

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

    def populate_list_store(self):
        self.list_store.clear()
        member = Member()
        memberdboperation = getAdapter(member, IDbOperation)
        members = memberdboperation.get()
```

```
        for member in members:
            number = member.number
            name = member.name
            self.list_store.append((member, number, name,))

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

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

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

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

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

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

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

    def on_update_clicked(self, *args):
        number = self.number.get_text()
        name = self.name.get_text()
        treeselection = self.treeview.get_selection()
        model, iter = treeselection.get_selected()
        if not iter:
```

```
            return
        member = self.list_store.get_value(iter, 0)
        member.number = number
        member.name = name
        self.update(member)
        self.list_store.set(iter, 1, number, 2, name)

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

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

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

memberwindow = MemberWindow()
```

Модуль *components.py*:

```
from zope.interface import implements

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

class Book(object):

    implements(IBook)

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

class Member(object):

    implements(IMember)
```

```
    number = ""
    name = ""

class Circulation(object):

    implements(ICirculation)

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

Модуль *interfaces.py*:

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


class IBook(Interface):

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


class IMember(Interface):

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


class ICirculation(Interface):

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


class IRelationalDatabase(Interface):

    def commit():
        pass

    def rollback():
        pass

    def cursor():
        pass
```

```
    def get_next_id():
        pass


class IObjectDatabase(Interface):

    def commit():
        pass

    def rollback():
        pass

    def container():
        pass

    def get_next_id():
        pass


class IDbOperation(Interface):

    def get():
        pass

    def add():
        pass

    def update():
        pass

    def delete():
        pass
```

Модуль *member.py*:

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

from components import Member

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

```
class MemberRDbOperation(object):

    implements(IDbOperation)
    adapts(IMember)

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

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

    def add(self):
        db = getUtility(IRelationalDatabase)
        cr = db.cursor()
        next_id = db.get_next_id("members")
```

```
        number = self.member.number
        name = self.member.name
        cr.execute("""INSERT INTO members
                        (id, number, name)
                      VALUES (?, ?, ?)""",
                   (next_id, number, name))
        cr.close()
        db.commit()
        self.member.id = next_id

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

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


class MemberODbOperation(object):

    implements(IDbOperation)
    adapts(IMember)

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

    def get(self):
```

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

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

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

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

9.4. PySQLite

9.5. ZODB

Глава 10. Сервер Интернет-приложений Pyramid

Среда разработки Интернет-приложений Pyramid использует регистратор компонент, встроенный в ZCA, в качестве регистра компонент разрабатываемого приложения. The Zope Component Architecture is referred to colloquially as the ”ZCA.”

The zope.component API used to access data in a traditional Zope application can be opaque. For example, here is a typical ”unnamed utility”lookup using the zope.component.getUtility() global API as it might appear in a traditional Zope application:

1 2 3

from pyramid.interfaces import ISettings from zope.component import getUtility settings = getUtility(ISettings)

After this code runs, settings will be a Python dictionary. But it’s unlikely that any ”civilian”will be able to figure this out just by reading the code casually. When the zope.component.getUtility API is used by a developer, the conceptual load on a casual reader of code is high.

While the ZCA is an excellent tool with which to build a framework such as Pyramid, it is not always the best tool with which to build an application due to the opacity of the zope.component APIs. Accordingly, Pyramid tends to hide the presence of the ZCA from application developers. You needn’t understand the ZCA to create a Pyramid application; its use is effectively only a framework implementation detail.

However, developers who are already used to writing Zope applications often still wish to use the ZCA while building a Pyramid application. Pyramid makes this possible. Using the ZCA global API in a Pyramid application

Zope uses a single ZCA registry—the ”global”ZCA registry—for all Zope applications that run in the same Python process, effectively making it impossible to run more than one Zope application in a single process.

However, for ease of deployment, it’s often useful to be able to run more than a single application per process. For example, use of a PasteDeploy ”composite”allows you to run separate individual WSGI applications in the same process, each answering requests for some URL prefix. This makes it possible to run, for example, a TurboGears application at /turbogears and a Pyramid application at /pyramid, both served up using the same WSGI server within a single Python process.

Most production Zope applications are relatively large, making it impractical due to memory constraints to run more than one Zope application per Python process. However, a Pyramid application may be very small and consume very little memory, so it’s a reasonable goal to be able to run more than one Pyramid application per process.

In order to make it possible to run more than one Pyramid application in a

single process, Pyramid defaults to using a separate ZCA registry per application.

While this services a reasonable goal, it causes some issues when trying to use patterns which you might use to build a typical Zope application to build a Pyramid application. Without special help, ZCA "global"APIs such as zope.component.getUtilit and zope.component.getSiteManager() will use the ZCA "global"registry. Therefore, these APIs will appear to fail when used in a Pyramid application, because they'll be consulting the ZCA global registry rather than the component registry associated with your Pyramid application.

There are three ways to fix this: by disusing the ZCA global API entirely, by using pyramid.config.Configurator.hook$_z ca()or bypassing the ZCA global registry to the Co$r

ZCA "global"API functions such as zope.component.getSiteManager, zope.component.g zope.component.getAdapter(), and zope.component.getMultiAdapter() aren't strictly necessary. Every component registry has a method API that offers the same functionality; it can be used instead. For example, presuming the registry value below is a Zope Component Architecture component registry, the following bit of code is equivalent to zope.component.getUtility(IFoo):

registry.getUtility(IFoo)

The full method API is documented in the zope.component package, but it largely mirrors the "global"API almost exactly.

If you are willing to disuse the "global"ZCA APIs and use the method interface of a registry instead, you need only know how to obtain the Pyramid component registry.

There are two ways of doing so:

use the pyramid.threadlocal.get$_c urrent_r egistry() function within Pyramid view or re

See Thread Locals for more information about pyramid.threadlocal.get$_c urrent_r egistry(

Consider the following bit of idiomatic Pyramid startup code:

1 2 3 4 5 6

```
from pyramid.config import Configurator
def app(global_settings, **settings) : config = Configurator(settings =
settings)config.include(' some.other.package ')returnconfig.make_wsgi_app()
```

When the app function above is run, a Configurator is constructed. When the configurator is created, it creates a new application registry (a ZCA component registry). A new registry is constructed whenever the registry argument is omitted, when a Configurator constructor is called, or when a registry argument with a value of None is passed to a Configurator constructor.

During a request, the application registry created by the Configurator is "made current". This means calls to get$_c urrent_r egistry() in the thread handling the request v

As a result, application developers can use get$_c urrent_r egistry to get the registry and th

To "fix"this and make the ZCA global APIs use the "current"Pyramid registry, you need to call hook$_z ca() within your setup code. For example :

1 2 3 4 5 6 7

from pyramid.config import Configurator

def app(global$_settings, **settings) : config = Configurator(settings = settings)config.hook_zca()config.include('some.other.application')returnconfig.ma$

We've added a line to our original startup code, line number 5, which calls config.hook$_zca()$.$The effect of this line under the hood is that an analogue of the following$

1 2 3

from zope.component import getSiteManager from pyramid.threadlocal import get$_current_registrygetSiteManager.sethook(get_current_registry)$

This causes the ZCA global API to start using the Pyramid application registry in threads which are running a Pyramid request.

Calling hook$_z ca is usually sufficient to "fix" the problem of being able to use the glob$

You can tell your Pyramid application to use the ZCA global registry at startup time instead of constructing a new one:

1 2 3 4 5 6 7 8 9

from zope.component import getGlobalSiteManager from pyramid.config import Configurator

def app(global$_settings, **settings) : globalreg = getGlobalSiteManager()config$ $Configurator(registry = globalreg)config.setup_registry(settings = settings)confi$

Lines 5, 6, and 7 above are the interesting ones. Line 5 retrieves the global ZCA component registry. Line 6 creates a Configurator, passing the global ZCA registry into its constructor as the registry argument. Line 7 "sets up" the global registry with Pyramid-specific registrations; this is code that is normally executed when a registry is constructed rather than created, but we must call it "by hand" when we pass an explicit registry.

At this point, Pyramid will use the ZCA global registry rather than creating a new application-specific registry. Since by default the ZCA global API will use this registry, things will work as you might expect in a Zope app when you use the global ZCA API.

10.1. Заключение

Глава 11. Методические указания к выполнению лабораторных работ

Чтобы знания, полученные в предыдущих частях учебного пособия, создали почву для формирования навыков использования компонентной архитектуры необходимо выполнить ряд лабораторных работ, которые представлены в разделах данной главы. Задания на лабораторные работы сформированы таким образом, чтобы следующая лабораторная работа дополняла материал предыдущей. Это позволяет смотреть на процесс обучения как на маленький проект, состоящий из четырех этапов: разработки компоненты, моделирующей предметную некоторую область или известную структуру данных; создание адаптера интерфейса компоненты из первой лабораторной работы к одной из компонент, представленных в другом варианте задания; разработка оконно интерфейса пользователя, позволяющая вносить изменения в данные компоненты через ее интерфейс; реализация распределенной версии приложения при помощи прокси-адаптера.

11.1. Разработка и тестирование компоненты

Варианты задания

11.2. Реализация адаптера интерфейса

Варианты задания

11.3. Создание интерфейса пользователя

Варианты задания

11.4. Моделирование распределенного приложения

Варианты задания

Глава 12. Справочник

adaptedBy

Функция возвращает перечень интерфейсов, адаптируемых объектом-адаптером.

- Модуль: `zope.component`
- Сигнатура: *adaptedBy(object)*

Пример:

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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

adapter

Адаптерами выступают любые вызываемые (callable) объекты. Также существует специальный декоратор *adapter*, который позволяет декларировать вызываемые объекты как адаптеры для указанных интерфейсов (или классов).

- Модуль: `zope.component`
- Спецификация: *adapter(*interfaces)*

Пример:

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

```
    from zope.component import adapter
    from zope.interface import implements

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

    class Job(object):
        implements(IJob)

    class IPerson(Interface):

        name = Attribute("Name")
        job = Attribute("Job")

    class Person(object):
        implements(IPerson)

        name = None
        job = None

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

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

adapts

Функция определяющая, что некоторый класс является адаптером.

- Модуль: `zope.component`
- Спецификация: *adapts(*interfaces)*

Пример:

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

    class FrontDeskNG(object):
```

```
    implements(IDesk)
    adapts(IGuest)

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

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

alsoProvides

Задает объекту перечень интерфейсов, которые он обслуживать, в дополнение к ранее декларированным. Параметры функции, следующие за объектом, – перечень обслуживаемых интерфейсов (один или несколько).

- Модуль: `zope.interface`
- Спецификация: *alsoProvides(object, *interfaces)*

Пример:

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

class IPerson(Interface):

    name = Attribute("Name of person")

class IStudent(Interface):

    college = Attribute("Name of college")

class Person(object):

    implements(IDesk)
    name = u""
```

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

You can test it like this:

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

Attribute

Класс, предназначенный для определения атрибутов в интерфейсе.

- Модуль: `zope.interface`
- Спецификация: *Attribute(name, doc="")*
- Смотри также: Interface

Пример:

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

class IPerson(Interface):

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

classImplements

Задает дополнительные интерфейсы классу, которые он в системе будет реализовывать. Перечень аргументов, следующих за классом – задаваемые интерфейсы, которые добавляются к списку ранее заданных интерфейсов.

- Модуль: `zope.interface`
- Спецификация: *classImplements(cls, *interfaces)*

Пример:

```
    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("Имя человека")

    class IStudent(IPerson):

        college = Attribute("Название колледжа")

    class Person(object):

        implements(IDesk)
        name = u""
        college = u""

    classImplements(Person, IStudent)
    ganes = Person()
    ganes.name = "Баролби Гейнс"
    ganes.college = "Саус-хермондский институт технологий"
```

Результат тестируется следующим образом:

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

classImplementsOnly

Задает классу набор реализуемых им интерфейсов. Перечень аргументов после класса – задаваемые интерфейсы. Ранее декларированные интерфейсы заменяются данным перечнем.

- Модуль: `zope.interface`
- Спецификация: *classImplementsOnly(cls, *interfaces)*

Пример:

```
    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"
```

Результат тестируется следующим образом:

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

classProvides

Обычно если класс реализует определенный интерфейс, то его экземпляры обслуживают этот интерфейс. Но когда надо, чтобы именно класс обслуживал некоторый интерфейс, то это реализуется функцией `classProvides`.

- Модуль: `zope.interface`
- Спецификация: *classProvides(*interfaces)*

Пример:

```
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"
```

Результат тестируется следующим образом:

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

ComponentLookupError

Исключение, которое инициируется, если поиск компоненты не удался. Пример:

```
    class IPerson(Interface):

        name = Attribute("Name of person")

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

ComponentLookupError: ...
```

createObject

Создает объект, используя фабрику.

Производит поиск поименованной фабрики в текущем регистре компонент (менеджере сайта) и запускает ее с заданными аргументами. Если такой фабрики не удается найти, инициируется исключение `ComponentLookupError`. Возвращает созданный объект.

При помощи ключевого слова *context* в аргументе функции задается контекст, где следует искать фабрику, в отличие от основного регистра. Ограничения технологии не позволяют передавать аргумент по ключу "context"в фабрику.

- Модуль: `zope.component`
- Спецификация: *createObject(factory_name, *args, **kwargs)*

Пример:

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

    class IDatabase(Interface):

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

    class FakeDb(object):

        implements(IDatabase)

        def getConnection(self):
            return "connection"

    from zope.component.factory import Factory

    factory = Factory(FakeDb, 'FakeDb')

    from zope.component import getGlobalSiteManager
    gsm = getGlobalSiteManager()

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

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

Declaration

Непосредственно не запускается.

directlyProvidedBy

Возвращает список интерфейсов, которые непосредственно обслуживаются объектом, переданным в параметре.

- Модуль: `zope.interface`
- Спецификация: *directlyProvidedBy(object)*

Пример:

```
    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
```

directlyProvides

Задает перечень интерфейсов, обслуживаемых объектом. Аргументы, непосредственно следующие за объектом – перечень декларируемых интерфейсов. Заданные интерфейсы заменяют декларированные ранее.

- Модуль: `zope.interface`
- Спецификация: *directlyProvides(object, *interfaces)*

Пример:

```
    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
```

getAdapter

Возвращает поименованный адаптер к заданному интерфейсу для заданного объекта. Если подходящего адаптера не получается найти, создается исключительная ситуация `ComponentLookupError`.

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

Пример:

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

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

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


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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

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

    class Guest(object):

        implements(IGuest)

        def __init__(self, name, place):
            self.name = name
            self.place = place

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

    IDesk.providedBy(jack_frontdesk)
True

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

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

getAdapterInContext

Вместо этой функции следует использовать аргумент *context* функции getAdapter.

- Модуль: `zope.component`
- Спецификация: *getAdapterInContext(object, interface, context)*
- Смотри также: queryAdapterInContext

Пример:

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

    class Context(object):
        def __init__(self, sm):
```

```
        self.sm = sm
    def __conform__(self, interface):
        if interface.isOrExtends(IComponentLookup):
            return self.sm

context = Context(sm)

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

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

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


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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

class Guest(object):

    implements(IGuest)

    def __init__(self, name, place):
        self.name = name
        self.place = place

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

```
    jack_frontdesk = FrontDeskNG(jack)

    IDesk.providedBy(jack_frontdesk)
True

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

    from zope.component import getAdapterInContext

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

getAdapters

Осуществляет поиск всех подходящих адаптеров к заданному интерфейсу и заданным объектам. Возвращает список адаптеров. Если в списке есть поименованный адаптер, то выдается наиболее специфический адаптер с заданным именем.

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

Пример:

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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

```
        }

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

    from zope.component import getGlobalSiteManager
    gsm = getGlobalSiteManager()

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

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

getAllUtilitiesRegisteredFor

Возвращает все зарегистрированные утилиты для заданного интерфейса, в том числе и те, которые были заменены (overridden). По возвращаемому значению можно осуществлять итерацию, где каждый элемент – это утилита.

- Модуль: `zope.component`
- Спецификация: *getAllUtilitiesRegisteredFor(interface)*

Пример:

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

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

    class Greeter(object):

        implements(IGreeter)

        def greet(self, name):
            print "Hello", name

    from zope.component import getGlobalSiteManager
    gsm = getGlobalSiteManager()

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

```
    from zope.component import getAllUtilitiesRegisteredFor

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

getFactoriesFor

Возвращает список двоек (name, factory) зарегистрированных фабрик классов, реализующих заданный интерфейс.

- Модуль: `zope.component`
- Спецификация: *getFactoriesFor(interface, context=None)*

Пример:

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

getFactoryInterfaces

Возвращает перечень интерфейсов, реализуемых фабрикой. Производит поиск по заданному имени фабрики, которая находится в ближайшем контексте, затем возвращает либо ее интерфейс, либо n-ку (tuple) интерфейсов, которые будут обслуживать объекты, создаваемые данной фабрикой.

- Модуль: `zope.component`
- Спецификация: *getFactoryInterfaces(name, context=None)*

Пример:

```
    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>
```

getGlobalSiteManager

Функция возвращает глобальный регистр компонент (глобальный менеджер сайта). Эта функция всегда должна успешно выполнится. Возвращаемый регистр обслуживает интерфейс *IGlobalSiteManager*.

- Модуль: `zope.component`
- Спецификация: *getGlobalSiteManager()*

Пример:

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

getMultiAdapter

Осуществляет поиск мультиадаптера (multi-adapter) к заданному интерфейсу и набору адаптируемых объектов. Возвращает мультиадаптер, который способен адаптировать эти объекты к указанному интерфейсу. Если такого адаптера найти не удается, то создается исключение *ComponentLookupError*. Пустое имя адаптера зарезервировано для непоименнованных адаптеров. Функции, реализующие поиск непоименнованных адаптеров, – это функции поиска адаптеров, где в качестве имени адаптера задается пустая строка.

- Модуль: `zope.component`
- Спецификация: *getMultiAdapter(objects, interface=Interface, name=”, context=None)*
- Смотри также: queryMultiAdapter

Пример:

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

class IAdapteeOne(Interface):
    pass

class IAdapteeTwo(Interface):
    pass
```

```
    class IFunctionality(Interface):
        pass

    class MyFunctionality(object):
        implements(IFunctionality)
        adapts(IAdapteeOne, IAdapteeTwo)

        def __init__(self, one, two):
            self.one = one
            self.two = two

    from zope.component import getGlobalSiteManager
    gsm = getGlobalSiteManager()

    gsm.registerAdapter(MyFunctionality)

    class One(object):
        implements(IAdapteeOne)

    class Two(object):
        implements(IAdapteeTwo)

    one = One()
    two = Two()

    from zope.component import getMultiAdapter

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

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

getSiteManager

Возвращает регистр компонент (менеджер сайта), который является наиближайщем в заданном контексте. Если параметр *context* равен *None*, то возвращается глобальный регистр. Если параметр *context* не равен *None*, то предполагается, что найдется какой-либо адаптер из *context*-а *IComponentLookup*.

Если же найти такой адаптер на самом деле не удалось, создается исключение *ComponentLookupError*.

- Модуль: `zope.component`
- Спецификация: *getSiteManager(context=None)*

Пример 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
```

Пример 2:

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

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

getUtilitiesFor

Осуществляет поиск зарегистрированных утилит, которые обслуживают заданный интерфейс. Возвращает объект, генерирующий двойки (name, utility).

- Модуль: `zope.component`

- Спецификация: *getUtilitiesFor(interface)*

  Пример:

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

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

class Greeter(object):

    implements(IGreeter)

    def greet(self, name):
        print "Hello", name

from zope.component import getGlobalSiteManager
gsm = getGlobalSiteManager()

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

from zope.component import getUtilitiesFor

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

getUtility

Возвращает утилиту, которая обслуживает заданный интерфейс. Возвращает утилиту, которая находится в ближайшем из контекстов. Если такой утилиты не удается найти, то создается исключение `ComponentLookupError`.

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

  Пример:

```
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'
```

handle

Запускает все обработчики (handlers) для заданных объектов. Обработчики – это фабрики адаптеров-подписчиков (subscription adapter), которые ничего не возвращают. Они просто выполняют некоторую процедуру. Обработчики обычно используются для реализации реакции на событие.

- Модуль: `zope.component`
- Спецификация: *handle(*objects)*

Пример:

```
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'
```

implementedBy

Возвращает перечень интерфейсов, реализуемых заданным классом.

- Модуль: `zope.interface`
- Спецификация: *implementedBy(class_)*

Пример 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>
```

Пример 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']
```

implementer

Создает декоратор (decorator) для задания интерфейса, реализуемого фабрикой. Возвращает исполняемый объект (callable), которые представляет собой фабрику, обрамляющую объект, переданный декоратору в качестве параметра.

- Модуль: `zope.interface`
- Спецификация: *implementer(*interfaces)*

Пример:

```
    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>]
```

implements

Функция вызывается внутри определения класса. Функция задает перечень интерфейсов, которые класс намеревается реализовать. Реализуемые интерфейсы задаются в качестве параметров функции. Заданные интерфейсы добавляются ранее декларированным, включая интерфейсы, получаемые при наследовании. Функция *implementsOnly* используется для задания перечня интерфейсов, при этом унаследованные и ранее декларированные интерфейсы заменяются.

- Модуль: `zope.interface`
- Спецификация: *implements(*interfaces)*

Пример:

```
    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
```

implementsOnly

Задает перечень интерфейсов, реализуемых классом, при этом ранее сделанные и унаследованные декларации интерфейсов отменяются (overridden). Эта функция выполняется в процессе определения класса. Аргументы функции – декларируемые интерфейсы.

- Модуль: `zope.interface`
- Спецификация: *implementsOnly(*interfaces)*

Пример:

```
    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
```

Interface

При помощи данного класса задаются интерфейсы. Интерфейсы – это особые классы Python. Чтобы определить класс-интерфейс надо просто унаследовать `Interface`.

- Модуль: `zope.interface`
- Спецификация: *Interface(name, doc="")*

Пример 1:

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

class IPerson(Interface):

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

Пример 2:

```
from zope.interface import Interface

class IHost(Interface):

    def goodmorning(guest):
        """Say good morning to guest"""
```

moduleProvides

Определить модуль Python как объект, обслуживающий заданный интерфейс. Функция используется в тексте определения модуля. Аргументы –

один или несколько интерфейсов. Исключительная ситуация создается в случае, если в модуле уже ранее встречалась такая декларация. Другими словами, в модуле декларация интерфейсов должна быть только одна.

Функция *moduleProvides* – это удобный вариант функции `directlyProvides`, применяемой к объекту-модулю.

- Модуль: `zope.interface`
- Спецификация: *moduleProvides(*interfaces)*
- Смотри также: directlyProvides

В исходном коде модуля *zope.component* эта функция применяется, а именно, в файле *__init__.py* есть следующие операторы:

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

Таким образом, *zope.component* обслуживает два интерфейса: *IComponentArchitectu* и *IComponentRegistrationConvenience*.

noLongerProvides

Исключить заданный интерфейс из списка интерфейсов, обслуживаемых объектом.

- Модуль: `zope.interface`
- Спецификация: *noLongerProvides(object, interface)*

Пример:

```
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
```

provideAdapter

Вместо этой функции рекомендовано использовать функцию registerAdapter
.

provideHandler

Вместо этой функции рекомендовано использовать функцию registerHandler
.

provideSubscriptionAdapter

Вместо этой функции рекомендовано использовать функцию registerSubscriptionAd
.

provideUtility

Вместо этой функции рекомендовано использовать функцию registerUtility
.

providedBy

Функция проверят, обслуживает ли объект указанный интерфейс. Возвращается *истина*, если это так, в том числе, если объект обслуживает этот интерфейс через наследование интерфейсов.

- Модуль: `zope.interface`
- Спецификация: *providedBy(object)*

Пример 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"
```

Результат проверяется следующим образом:

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

Пример 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"
```

Следующий код позволяет получить полный перечень интерфейсов, обслуживаемых объектом::

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

queryAdapter

Производит поиск адаптера к заданному интерфейсу для заданного объекта. Возвращает подходящий адаптер, если такого адаптера нет, то возвратить значение, заданное параметром *default*.

- Модуль: `zope.component`
- Спецификация: *queryAdapter(object, interface=Interface, name=u", default=None, context=None)*

Пример:

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

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

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

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

    class FrontDeskNG(object):

        implements(IDesk)
        adapts(IGuest)

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

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

    class Guest(object):

        implements(IGuest)

        def __init__(self, name, place):
            self.name = name
            self.place = place

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

    IDesk.providedBy(jack_frontdesk)
True

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

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

queryAdapterInContext

Осуществляет поиск адаптера специального вида к заданному интерфейсу для заданного объекта.

NOTE: Этот метод должен вызываться только в специальных случаях, когда по каким-либо причинам следует изменить обычный механизм получения доступа к компонентам. Иначе следует применять функцию *interface*, как в этом примере:

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

Функция возвращает адаптер, который способен адаптировать заданный объект к заданному интерфейсу. Если такого адаптера не удается найти, то возвращается значение, указанное в параметре *default*.

Контекст адаптируется к интерфейсу *IServiceService*, полученный адаптер используется как сервис.

Если объект, переданный в параметр, включает метод *__conform__*, этот метод вызывается с запрашиваемым интерфейсом в качестве параметра. Если метод возвращает значение, отличное от *None*, это значение и возвращается. Иначе, если объект и так реализует заданный интерфейс, то он сам и будет возвращен.

- Модуль: `zope.component`
- Спецификация: *queryAdapterInContext(object, interface, context, default=None)*
- Смотри также: getAdapterInContext

Пример:

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

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

context = Context(sm)

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

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

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


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

    class FrontDeskNG(object):

        implements(IDesk)
        adapts(IGuest)

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

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

    class Guest(object):

        implements(IGuest)

        def __init__(self, name, place):
            self.name = name
            self.place = place

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

    IDesk.providedBy(jack_frontdesk)
True

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

```
    from zope.component import queryAdapterInContext

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

queryMultiAdapter

Осуществляет поиск мультиадаптера (multi-adapter) к заданному интерфейсу и набору адаптируемых объектов. Возвращает мультиадаптер, который способен адаптировать эти объекты к указанному интерфейсу. Если такого адаптера найти не удается, возвращается значение, переданное в параметре *default*. Пустое имя адаптера зарезервировано для непоименнованных адаптеров. Функции, реализующие поиск непоименнованных адаптеров, – это функции поиска адаптеров, где в качестве имени адаптера задается пустая строка.

- Модуль: `zope.component`
- Спецификация: *queryMultiAdapter(objects, interface=Interface, name=u'', default=None, context=None)*
- Смотри также: getMultiAdapter

Пример:

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

class IAdapteeOne(Interface):
    pass

class IAdapteeTwo(Interface):
    pass

class IFunctionality(Interface):
    pass

class MyFunctionality(object):
    implements(IFunctionality)
    adapts(IAdapteeOne, IAdapteeTwo)

    def __init__(self, one, two):
```

```
            self.one = one
            self.two = two

    from zope.component import getGlobalSiteManager
    gsm = getGlobalSiteManager()

    gsm.registerAdapter(MyFunctionality)

    class One(object):
        implements(IAdapteeOne)

    class Two(object):
        implements(IAdapteeTwo)

    one = One()
    two = Two()

    from zope.component import queryMultiAdapter

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

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

queryUtility

Функция используется для поиска утилиты, которая обслуживает заданный интерфейс. Если такой утилиты не удается найти, то возвращается значение, переданное в параметре *default*.

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

Пример:

```
    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'
```

registerAdapter

Эта функция используется для регистрации фабрики адаптеров.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registerAdapter(factory, required=None, provided=None, name=u'', info=u'')*
- Смотри также: unregisterAdapter

Пример:

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

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

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


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

```
    class FrontDeskNG(object):

        implements(IDesk)
        adapts(IGuest)

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

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

    class Guest(object):

        implements(IGuest)

        def __init__(self, name, place):
            self.name = name
            self.place = place

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

    IDesk.providedBy(jack_frontdesk)
True

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

Результат тестируется следующим образом:

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

registeredAdapters

Возвращает генератор *IAdapterRegistrations*, элементы которого описывают существующие зарегистрированные фабрики адаптеров.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registeredAdapters()*

Пример:

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

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

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


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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

class Guest(object):

    implements(IGuest)

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

```
            self.place = place

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

    IDesk.providedBy(jack_frontdesk)
True

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


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

registeredHandlers

Возвращает генератор *IHandlerRegistrations*, элементы которого описывают зарегистрированные обработчики.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registeredHandlers()*

Пример:

```
    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
```

registeredSubscriptionAdapters

Возвращает генератор *ISubscriptionAdapterRegistrations*, элементы которого описывают зарегистрированные адаптеры-подписчики.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registeredSubscriptionAdapters()*

Пример:

```
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
```

registeredUtilities

Эта функция возвращает генератор *IUtilityRegistrations*, элементы которого описывают зарегистрированные утилиты.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registeredUtilities()*

Пример:

```
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
```

registerHandler

Функция позволяет регистрировать обработчики (handlers). Обработчик – это адаптер-подписчик (subscription adapter), который ничего не возвращает, только выполняет заданную процедуру.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registerHandler(handler, required=None, name=u”, info=”)*
- Смотри также: unregisterHandler

Замечание: В современной реализации модуля `zope.component` пока нет поддержки поименованных подписчиков.

Пример:

```
    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'
```

registerSubscriptionAdapter

Функция используется для регистрации фабрики адаптеров-подписчиков (subscription adapter).

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registerSubscriptionAdapter(factory, required=None, provides=None, name=u'', info='')*
- Смотри также: unregisterSubscriptionAdapter

Пример:

```
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)
```

registerUtility

Функция позволяет зарегистрировать утилиту.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *registerUtility(component, provided=None, name=u”, info=u”)*
- Смотри также: unregisterUtility

Пример:

```
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)
```

subscribers

Функция позволяет получить перечень подписчиков (subscription adapter). Элементы перечня обслуживают заданный интерфейс (от которого они к тому же зависят). Перечень также определяется списком заданных объектов, переданных в функцию в качестве первого параметра.

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

Пример:

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

    doc = Document("A text\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']
```

unregisterAdapter

Функция используется для отмены регистрации фабрики адаптеров. Возвращается значение типа *boolean*. Если после выполнения операции состояние регистратора изменилось, то возвращается *истина*. Значение *ложь* возвращается, если параметр равен *None* или компонента не было зарегистрирована.

- Модуль: `zope.component - IComponentRegistry`

- Спецификация: *unregisterAdapter(factory=None, required=None, provided=None, name=u”)*
- Смотри также: registerAdapter

Пример:

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

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

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


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

class FrontDeskNG(object):

    implements(IDesk)
    adapts(IGuest)

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

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

class Guest(object):

    implements(IGuest)

    def __init__(self, name, place):
        self.name = name
        self.place = place

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

```
    IDesk.providedBy(jack_frontdesk)
True

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

Результат проверяется следующим образом:

```
    queryAdapter(jack, IDesk, 'ng6')
<FrontDeskNG object at ...>
```

Теперь отменяем регистрацию:

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

После отмены регистрации имеем:

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

unregisterHandler

Функция отменяет регистрацию обработчика (handler). Возвращается значение типа *boolean*. Если после выполнения операции состояние регистратора изменилось, то возвращается *истина*.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *unregisterHandler(handler=None, required=None, name=u'')*
- Смотри также: registerHandler

Пример:

```
    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 text\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
```

unregisterSubscriptionAdapter

Функция используется для отмены регистрации фабрики подписчиков (subscription adapter). Возвращается значение типа *boolean*. Если после выполнения операции состояние регистратора изменилось, то возвращается *истина*.

- Модуль: `zope.component - IComponentRegistry`
- Спецификация: *unregisterSubscriptionAdapter(factory=None, required=None, provides=None, name=u”)*
- Смотри также: registerSubscriptionAdapter

Пример:

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

Теперь отменяем регистрацию:

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

После отмены регистрации:

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

unregisterUtility

Функция используется для отмены регистрации утилиты. Возвращается значение типа *boolean*. Если после выполнения операции состояние регистратора изменилось, то возвращается *истина*. Значение *ложь* возвращается, если параметр равен *None* или утилита не была зарегистрирована.

- Модуль: `zope.component - IComponentRegistry`

- Спецификация: *unregisterUtility(component=None, provided=None, name=u'')*
- Смотри также: registerUtility

Пример:

```
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'
```

Теперь отменяем регистрацию:

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

После отмены регистрации имеем:

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

12.1. Современные компонентные архитектуры

Замечание об использовании интерфейсов в Go https://tour.golang.org/methods/5.

Учебное издание

Черкашин Евгений Александрович
Байджу Мутхукадан

Компонентное программирование в Python

ISBN 978-5-XXXXX-XXXX-X

Редактор *Е. А. Черкашин*
Верстка *Е. А. Черкашин*

Макет подготовлен при помощи системы LuaLaTeX

Темплан 2016 г. Поз. 186

Подписано в печать 28.12.2016. Формат 60×90 1/16.
Уч.-изд. л. 6,4. Усл. печ. л. 6,8. Тираж 100 экз. Заказ 170

Издательство ИРНИТУ
664074, г. Иркутск, ул. Лермонтова, 83