Design Patterns in PYTHON

Daniel Bracher, Isabella Dall'Oglio

1 Tasks

- Wählen Sie zumindest 5 verschiedenen Design-Pattern (zumindest 1 pro Kategorie) aus
- Ausgehend von einem UML-Klassendiagramm wird jedes Design-Pattern auf Umsetzung in Python untersucht.
- Erstellen Sie für jedes Pattern Beispiel-Code (inkl. Sphinx-Dokumentation)
- Erstellen Sie ein genaues Protokoll allen Aufzeichnungen
- Geben Sie das Protokoll und den Beispielcode für alle gewählten Design-Pattern ab

2 Aufteilung

- Ausgewählte Patterns:
    - Signalton Pattern -> I
    - Decorater Pattern -> I
    - Factory Pattern -> D
    - Iterator Pattern
    - Adapter Pattern -> D

3 Sphinx Dokumentation

Sphinx is a tool to create documentation for your projects, available in PyCharm. To install Sphinx for your Project, go to Settings -> Project Interpreter. Now click on the “Add”-Button and type in “Sphinx” and install it.
After installation you can choose Tools -> Sphinx Quickstart.
There are a couple of options you can now define on the command line.

- If you want to have a different Documentation-Directory.
- If you want to separate “Build” and “Source” directories

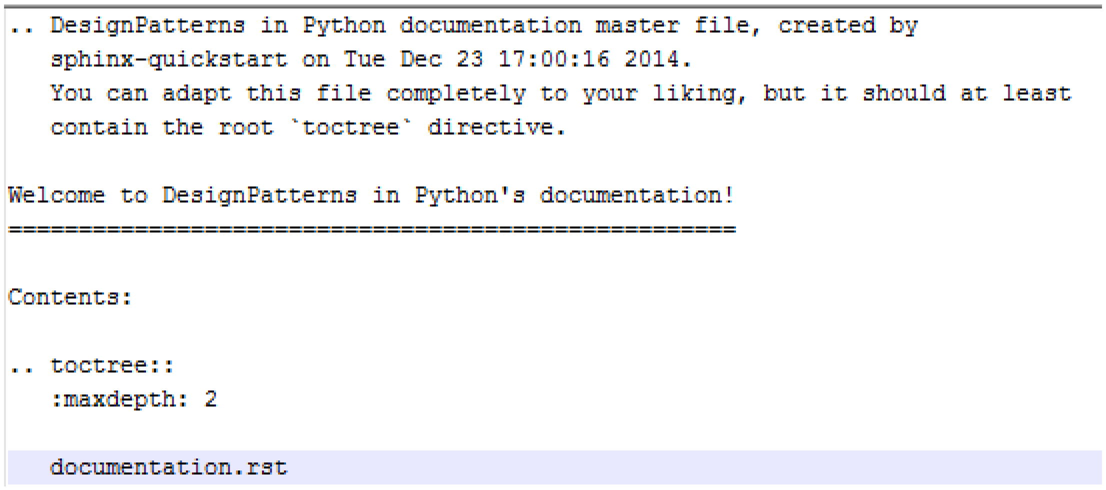
- Different settings for your project such as name, version, etc.
- If you want to use “.rst” or “.txt” files (use .rst for it is much easier to read)
- How you want to name your master document (index is fine)
- If you want to use different Sphinx-Extensions (coverage, autodoc, etc.)
- If you want to create a make and a batch file

Now that you have the basic structure, you still have to install sphinx itself. This works rather easy via “easy_install -U Sphinx”.

After the installation you must now provide a Configuration, that is runnable after each time you changed the code. To do so, go to Run - > Edit Configurations. Click on the + and choose Python docs -> Sphinx Task. Change the name to something like “SphinxDoc *Projectname*”. As Input set the Folder, where your make-File is (created by the Sphinx Quickstart). Create a Folder called “html” and set it as Output. Apply the changes and close the Edit-Config window.

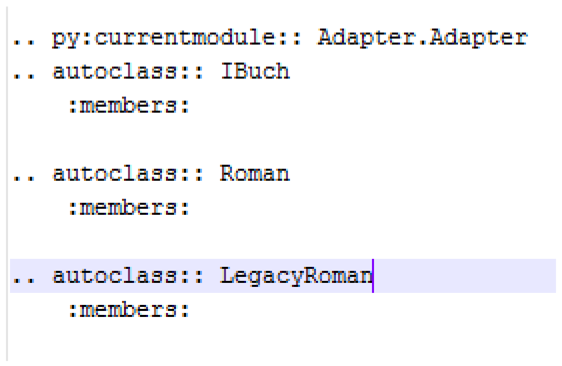
So now that you have set up the page, you might want to add the documentation of your classes. That works the following:

1) Open the index.rst file and write down the filename, you want to create your new subpage in



```
.. DesignPatterns in Python documentation master file, created by
   sphinx-quickstart on Tue Dec 23 17:00:16 2014.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to DesignPatterns in Python's documentation!
====================================================

Contents:

.. toctree::
   :maxdepth: 2

   documentation.rst
```

2) Then you create a file called like the one you wrote down in the index.rst file (in this example documentation.rst). Be aware, that the name has to be on the same indentation as the “:maxdepth: 2”.
3) In this file you can do headings (by underlining them with ===) and subheadings (by underlining them with ---).
But most importantly you can create the documentation of your classes. To do so write down the following, where Adapter.Adapter is *Module.Filename* and IBuch, Roman and LegacyRoman are the different class-names.



```
.. py:currentmodule:: Adapter.Adapter
.. autoclass:: IBuch
    :members:

.. autoclass:: Roman
    :members:

.. autoclass:: LegacyRoman
    :members:
```

4) The last thing you have got to do, is to run the previously set up configuration. To do so, choose the configuration in the upper right corner next to the run-button. Now you just click that button to run the script. (Do so every time you change something in your .rst files or your code).

4 “Interfaces” in Python

Interfaces in Python are actually non-existing. However, there is something like AbstractClasses, actually called AbstractBaseClasses (shortly ABC).

How it works

We start by defining a BaseClass (comparable to an Interface), and therein define the methods as abstract ones.

```python
import abc

class PluginBase(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source and return an object."""
        return
```

After doing so, we have to register the concrete classes as implementation of the base class.

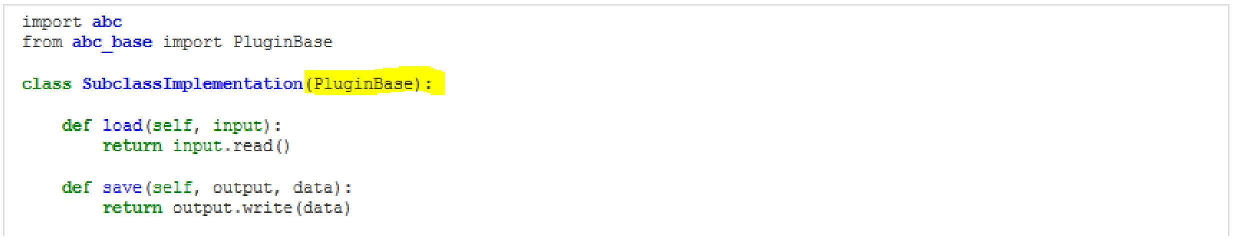
There are two ways to do so:

1) Register the class with the ABC

```python
import abc
from abc_base import PluginBase

class RegisteredImplementation(object):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)

PluginBase.register(RegisteredImplementation)
```

Benefit: All the abstract-methods have to be implemented

2) Subclass directly from the ABC



```
import abc
from abc_base import PluginBase

class SubclassImplementation(PluginBase):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)
```

[1]

5 Durchführung

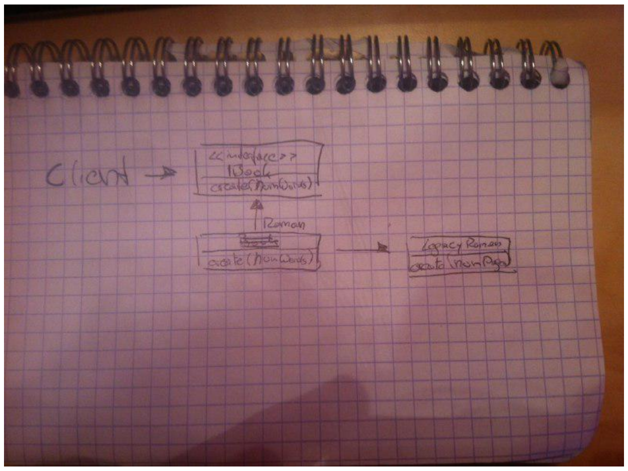
Adapter Pattern (Bracher)

When you have two interfaces, a problem that often (or sometimes) occurs, is that those two interfaces can’t communicate with each other. That’s why we need an Adapter-Pattern.
We could actually think of the adapter pattern in a more abstract way, like an adapter for your computer charger, that won’t fit into the wall, because you’re abroad. [2]

The way it actually works, is the following: The client calls the method of the adapter class that than redirects the call to the adapted interface. “This strategy can be implemented either with inheritance or with aggregation. Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.”

So for example, we have an Interface Book, with a function called “createBook()” which expects the number of pages as a parameter. Now we have a new customer, who doesn’t want to pass over the number of pages, but the number of words he has written.

So instead of modifying the class “Book” we use an AdapterClass, so the user can use the number of words he has written, and still create a book, which actually expects the number of pages.



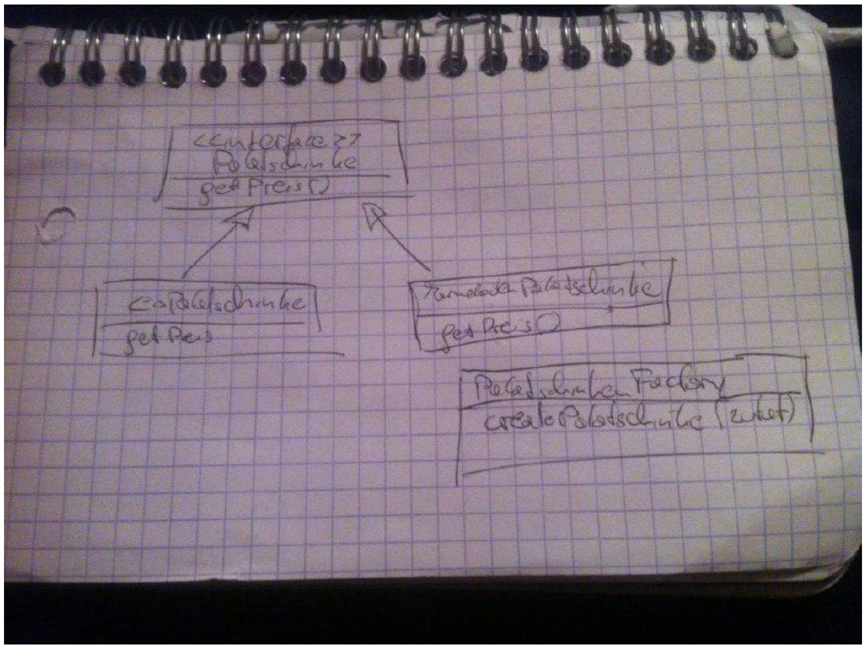
Factory Pattern (Bracher)

The Factory Pattern is all about creating new objects. The problem is that with *new* we always get a new object. We can't pass already existing objects or create objects of subclasses. We can't use more descriptive constructor names. So in other words, to expand the possibilities of the creation of objects, we define a factory method (aka "virtual constructor). [5] Another advantage of using the factory pattern is that we can chose at runtime which class to be instantiated.

How do we set up this pattern? We start with an interface of our products, for example Palatschinke with a method called getPrice(). Then we create two or three classes that implement this interface, such as MarmeladenPalatschinke and EisPalatschinke.

After that we create the very important PalatschinkenFactory, which is going to create our Palatschinken, through a static method called createPalatschinken(type), where type is either Eis or Marmelade.

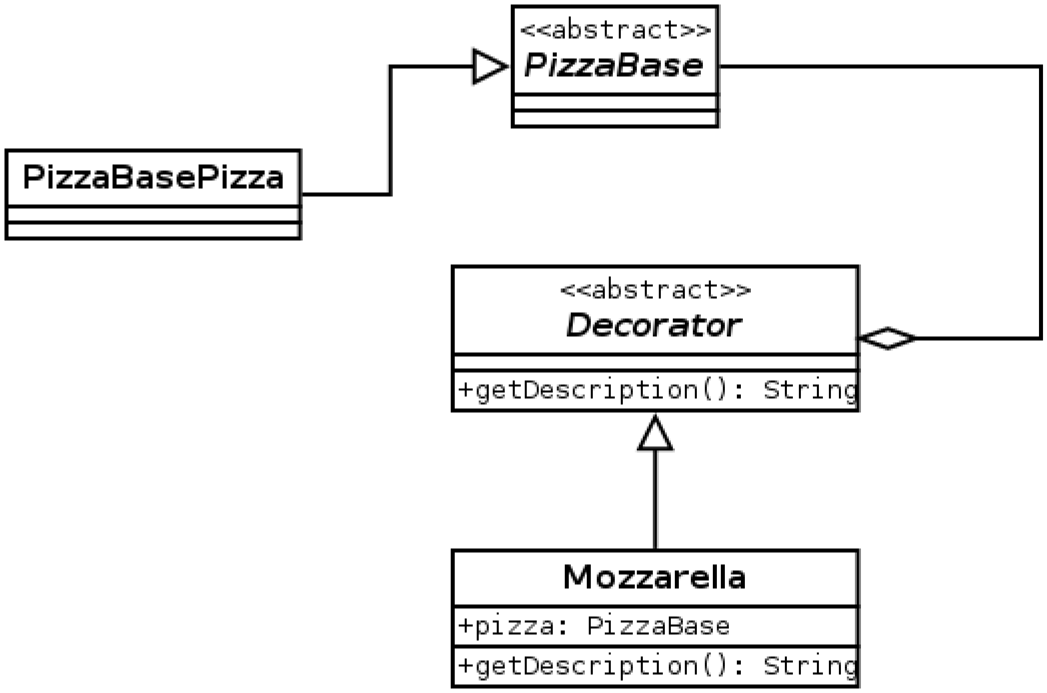
For another good and simple example, visit Aadesh Neupane's Site [6].



Decorator Pattern

In object-oriented programming, the decorator pattern is a design pattern that allows behaviour to be added to an existing object dynamically. The decorator pattern can be used to extend (decorate) the functionality of a certain object at run-time, independently of other instances of the same class, provided some groundwork is done at design time. [3]

UML:



Code:

```
class PizzaBase():
        """
        Stellt eine Pizza da.
        """
        @abstractmethod
        def getDescription(self):
        pass
```

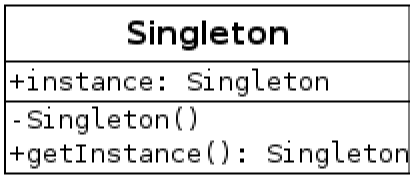
```
class Decorator(PizzaBase):
        """
        Der Decorator erbt von PizzaBase
        """

        pizza = PizzaBase()
        @abstractmethod
        def getDescription(self):
        pass

class Mozzarella(Decorator):
        pizza = ""

        def __init__(self,newpizza):
        self.pizza=self.getDescription(newpizza)

        def getDescription(self,newpizza):
        return  print("New Pizza with "+ newpizza + '+ Mozzarella')

class PizzaBasePizza(PizzaBase):
        """
        Diese Klasse verwendet PizzaBase
        """
        desc = ""

        def __init__(self, desc):
        self.desc = self.getDescription(desc)

        def getDescription(self):
        pass

        print('%s' % Mozzarella("Schinken "))
```

Signelton

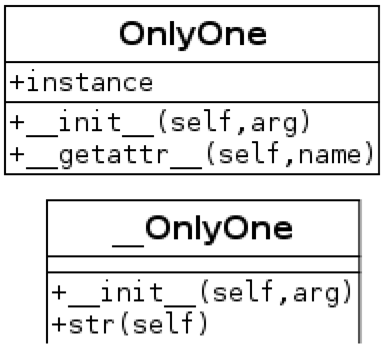
Possibly the simplest design pattern is the *singleton*, which is a way to provide one and only one object of a particular type. To accomplish this, you must take control of object creation out of the hands of the programmer. One convenient way to do this is to delegate to a single instance of a private nested inner class. [4]

The first time you create an OnlyOne, it initializes instance, but after that it just ignores you.

Allgemeines um UML:



UML zum Beispiel:



Code:

```
class OnlyOne:
    class __OnlyOne:

    """
    Der Konstruktor von der inneren Klasse __OnlyOne
    """
    def __init__(self, arg):
    self.val = arg
    def __str__(self):
    return repr(self) + self.val
    instance = None
```

```
        """
        Der Konstruktor der OnlyOne Klasse. Es wird geschaut ob OnlyOne eine Instanz hat,
        ha sie noch keine wird eine neue Instanz erzeugt.
        """
        def __init__(self, arg):
        if not OnlyOne.instance:
        OnlyOne.instance = OnlyOne.__OnlyOne(arg)
        else:
        OnlyOne.instance.val = arg
        """
        Diese Methode gibt ein Attribut zurück.
        """
        def __getattr__(self, name):
        return getattr(self.instance, name)

x = OnlyOne('sausage')
print(x)
y = OnlyOne('eggs')
print(y)
z = OnlyOne('spam')
print(z)
print(x)
print(y)
```

6 Quellen

[1] online; http://pymotw.com/2/abc/; zuletzt besucht: 21.12.2014
[2] online; http://www.vincehuston.org/dp/adapter.html; zuletzt besucht: 15.12.2014
[3] online;
http://stackoverflow.com/questions/8328824/whats-the-difference-between-python-decorators-and-decorator-pattern; zuletzt besucht: 22.12.2014
[4] online; http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Singleton.html
[5] online; http://www.vincehuston.org/dp/factory_method.html; zuletzt besucht: 23.12.2014
[6] online;
http://aadeshnpn.com.np/blog/2013/04/18/factory-method-design-pattern-in-pythonl/; zuletzt besucht: 23.12.2014