Krickl | Seckin

Design Pattern in Python

5BHITM

Design Pattern in Python

Inhalt

[Aufgabenstellung 2](#_Toc406961793)

[Zeitaufzeichnung 2](#_Toc406961794)

[Allgemein - Kategorien 3](#_Toc406961795)

[Factory Pattern 3](#_Toc406961796)

[UML 3](#_Toc406961797)

[Code 4](#_Toc406961798)

[Singelton Pattern 4](#_Toc406961799)

[UML 4](#_Toc406961800)

[Code ? 4](#_Toc406961801)

[Decorator Pattern 5](#_Toc406961802)

[UML 5](#_Toc406961803)

[Code ?? 6](#_Toc406961804)

[Command Pattern 7](#_Toc406961805)

[UML 7](#_Toc406961806)

[Code 7](#_Toc406961807)

[Strategy Pattern 7](#_Toc406961808)

[UML 7](#_Toc406961809)

[Code 7](#_Toc406961810)

[Quellen 7](#_Toc406961811)

# Aufgabenstellung

In einem Team (2) soll die These verifiziert oder falsifiziert werden.

* 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

Viel Erfolg!

Ressourcen:

http://www.aleax.it/gdd\_pydp.pdf

http://python-3-patterns-idioms-test.readthedocs.org/en/latest/index.html

http://legacy.python.org/workshops/1997-10/proceedings/savikko.html

head first: design pattern

# Zeitaufzeichnung

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Geschätzte Zeit in h** | **Tatsächliche Zeit in h** | **Verantwortung** |
| Recherchieren und nachschlagen | 1 |  | Krickl & Seckin |
| Factory UML | 0.2 | 0.1 | Krickl |
| Factory Code | 0.5 |  | Krickl |
| Singelton UML | 0.1 | 0.2 | Krickl |
| Singelton Code | 0.4 |  | Krickl |
| Decorator UML | 0.1 | 0.1 | Krickl |
| Decorator Code | 0.5 |  | Krickl |
| Strategy UML |  |  | Seckin |
| Strategy Code |  |  | Seckin |
| Command UML |  |  | Seckin |
| Command Code |  |  | Seckin |
| Protokoll erstellen | 2 | 2.5 | Krickl & Seckin |
|  |  |  |  |
| Summe Krickl | 4.8 |  |  |
| Summe Seckin |  |  |  |

# Allgemein - Kategorien

**Creational**: Beeinflusst die Art wie ein Objekt erstellt wird.

* Factory Pattern
* Singelton Pattern

**Structual**: Beeinflusst die Wege wie Objekte miteinander verbunden sind, damit bei einer Änderung des Systems nicht alle Verbindungen auch geändert werden müssen.

* Decorator Pattern

**Behavioral**: Beeinflusst Objekte, die viele verschiedene Aktionen durchführen in der Art wie sie diese Aktionen durchzuführen haben.

* Command Pattern
* Strategy Pattern

<http://python-3-patterns-idioms-test.readthedocs.org/en/latest/PatternConcept.html>

# Factory Pattern

Entkapselung der Objekterstellung

Wenn man mehrere verschiedene Datentypen in einer Klasse verwalten will, kann man das über Generizität lösen, das ist jedoch gefährlich, da man hier absolut jeden Datentypen einsetzen kann, auf den der Code nicht unbedingt abgestimmt ist. Um dieses Problem zu umgehen kann man das Factory Pattern verwenden. Es gibt eine Basisklasse die alles beinhaltet was jeder untergeordnete Typ können muss. Die Methoden dieser Basisklasse sind statisch.

http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Factory.html

## C:\Users\Asi\SCHULE3\SEW\factory.jpgUML

## Code

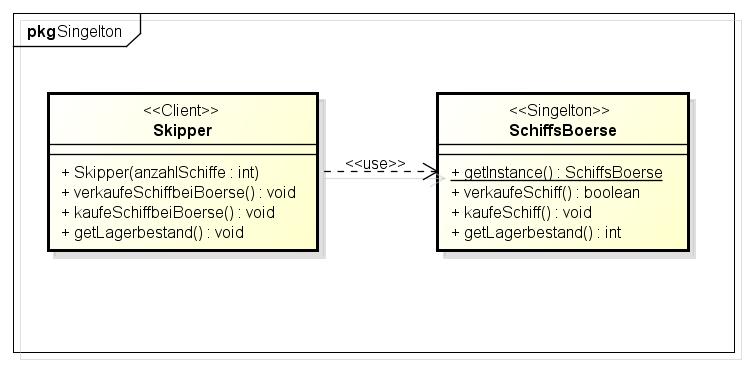
# Singelton Pattern

Wird verwendet um genau ein spezielles Objekt einer Klasse erstellen zu können. Zum Realisieren muss man aktiv in die Objekterstellung eingreifen und sie steuern. Oft setzt man das um indem man die Instanz des Objekts in eine innere Klasse delegiert.

Man kann sich das Singelton Pattern auch als eine Art statische Klasse vorstellen die aber intern nicht statisch ist.

http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Singleton.html

## UML



## Code ?

# Singleton/SingletonPattern.py

class OnlyOne:

class \_\_OnlyOne:

def \_\_init\_\_(self, arg):

self.val = arg

def \_\_str\_\_(self):

return repr(self) + self.val

instance = None

def \_\_init\_\_(self, arg):

if not OnlyOne.instance:

OnlyOne.instance = OnlyOne.\_\_OnlyOne(arg)

else:

OnlyOne.instance.val = arg

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)

print(`x`)

print(`y`)

print(`z`)

output = '''

<\_\_main\_\_.\_\_OnlyOne instance at 0076B7AC>sausage

<\_\_main\_\_.\_\_OnlyOne instance at 0076B7AC>eggs

<\_\_main\_\_.\_\_OnlyOne instance at 0076B7AC>spam

<\_\_main\_\_.\_\_OnlyOne instance at 0076B7AC>spam

<\_\_main\_\_.\_\_OnlyOne instance at 0076B7AC>spam

<\_\_main\_\_.OnlyOne instance at 0076C54C>

<\_\_main\_\_.OnlyOne instance at 0076DAAC>

<\_\_main\_\_.OnlyOne instance at 0076AA3C>

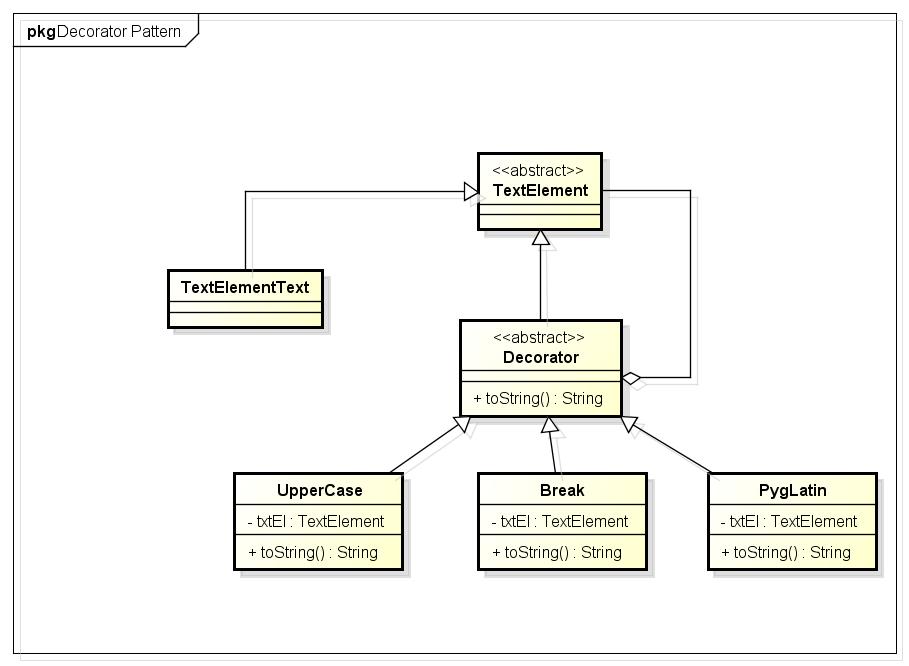
'''

# Decorator Pattern

Man kombiniert Objekte miteinander um ein Objekt zu erweitern. Es ist eine ähnliche Vorgehensweise wie mit Subklassen, jedoch aber findet die Anwendung des Decorator Patterns ohne Vererbung statt.  
Man kann viele Komponenten in der Laufzeit kombinieren wie man will, ohne eine aufwendige Subklassenhierarchie erstellen zu müssen.

http://www.pasteur.fr/formation/infobio/python/ch19s06.html

## UML



## Code ??

class A:

def \_\_init\_\_(self, b):

"""storing of the decoratee b (b is an instance of class B)"""

self.b = b

def \_\_getattr\_\_(self, name):

"""

methods/attributes A does not know about are delegated

to b

"""

return getattr(self.b, name)

class B:

def f(self):

return "result of f"

# Command Pattern

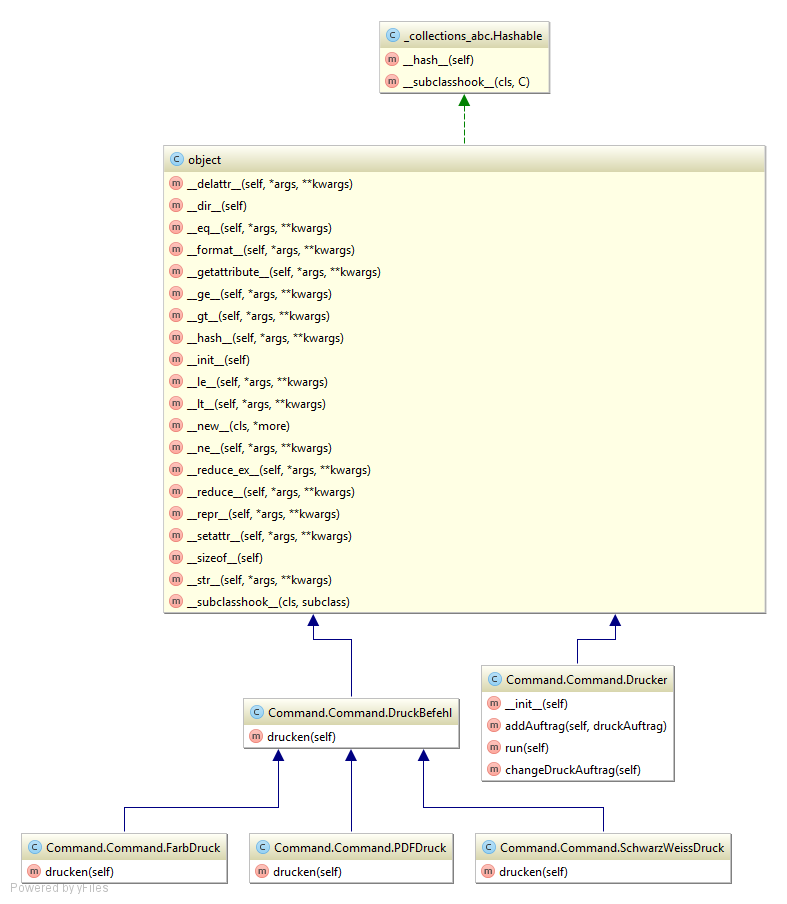
Ein Objekt wird verwendet um alle benötigten Informationen zu verkapseln und dann eine Methode aufrufen zu können. Mit dem Command-Design Pattern kann man bestimmte Aktionen einer Methode oder einem Objekt übergeben

=> Ein Befehl als Objekt kapseln:

Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren,

Operationen in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

## UML



## Code

\_\_author\_\_ **=** 'Berkay'

"""

Mit dem Command-Design Pattern kann man bestimmte Aktionen einer Methode oder einem Objekt übergeben

=> Ein Befehl als Objekt kapseln:

Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren,

Operationen in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

"""

# Sozuasgen das Interface

**class** **DruckBefehl:**

**def** drucken**(**self**):** **pass**

# konkrete Durchführung

**class** **SchwarzWeissDruck(**DruckBefehl**):**

**def** drucken**(**self**):**

"""

druckt Schwarz/Weiss aus

:return: \_string

"""

**print(**"==== Schwarz/Weiß gedruckt ===="**)**

# konkrete Durchführung

**class** **FarbDruck(**DruckBefehl**):**

**def** drucken**(**self**):**

"""

druckt FARBIG aus

:return \_string:

"""

**print(**"==== FARBIG gedruckt ===="**)**

# konkrete Durchführung

**class** **PDFDruck(**DruckBefehl**):**

**def** drucken**(**self**):**

"""

druckt PDF aus

:return: \_string

"""

**print(**"==== PDF - Druck ===="**)**

# An object that holds commands:

**class** **Drucker:**

**def** \_\_init\_\_**(**self**):**

"""

Beim Aufruf dieser Klasse

:rtype : object

:return: None

"""

self**.**druckAuftraege **=** **[]** # leeres Array

**def** addAuftrag**(**self**,** druckAuftrag**):**

"""

Fügt ein Druckauftrag zur Auftragsliste hinzu

:param druckAuftrag:

:return: None

"""

self**.**druckAuftraege**.**append**(**druckAuftrag**)** # dem Array hinzufügen

**def** run**(**self**):**

"""

Durchführung aller Druckaufträge

:return:

"""

**for** c **in** self**.**druckAuftraege**:**

c**.**drucken**()**

**def** changeDruckAuftrag**(**self**):**

**pass**

# Beispiel - Client

meinDrucker **=** Drucker**()**

meinDrucker**.**addAuftrag**(**SchwarzWeissDruck**())**

meinDrucker**.**addAuftrag**(**FarbDruck**())**

meinDrucker**.**addAuftrag**(**PDFDruck**())**

meinDrucker**.**run**()**

# Strategy Pattern

Das Strategy-Pattern wird verwendet um während der Laufzeit Änderungen durchführen zu können.

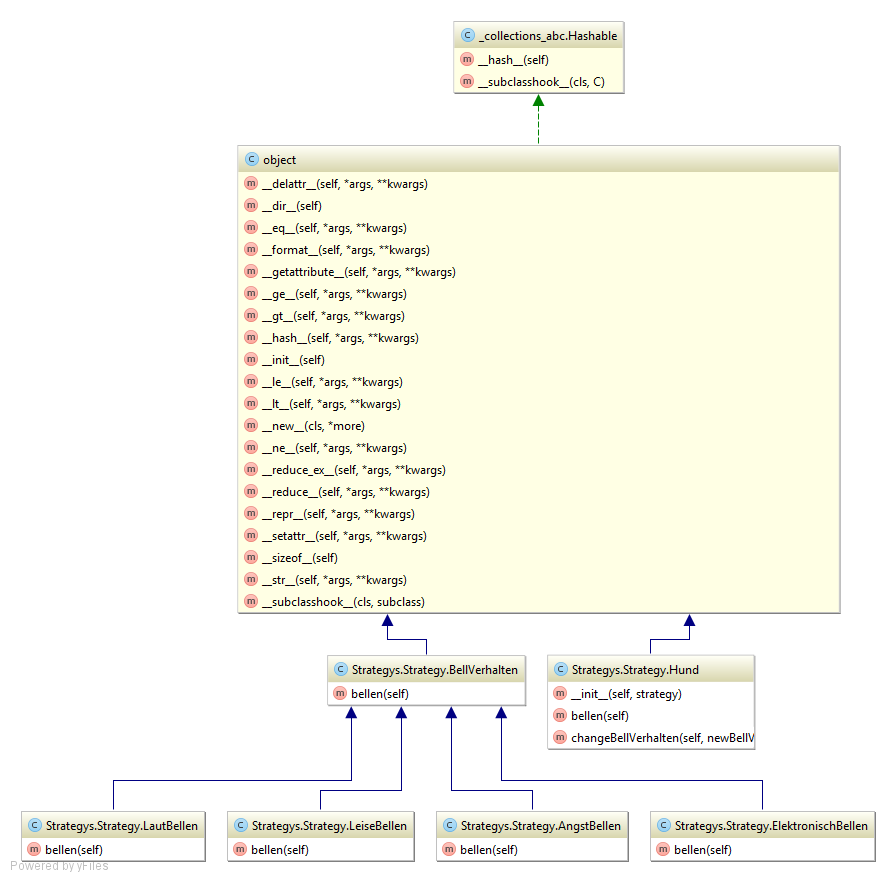
Man benötigt Interfaces, In Python aber gibt es diese nicht direkt, wie in Java. Jedoch gibt es als Abhilfe die Vererbung in Python(auch Mehrfach-Vererbung), welche folgende Syntax hat:

**class** **abgeleiteteKlasse(BasisKlasse):**

…  
 …

Dies wird verwendet für Attribut Referenzen: Wenn ein benötigtes Attribut nicht in dieser Klasse gefunden wird, wird in der Basis Klasse gesucht. In unserem Beispiel wird also das Verhalten(Funktion, Algorithmus) von dem Objekt Hund in eine eigene Strategyklasse ausgelagert.  
*Der Context(Hund-Klasse) hält eine Referenz auf sein Strategieobjekt und wenn er das ausgelagerte Verhalten ausführen soll, so delegiert er den Aufruf an sein referenziertes Strategieobjekt“ [3]*

## UML



## Code

\_\_author\_\_ **=** 'SECKIN Berkay'

# The strategy interface:

**class** **BellVerhalten** **:**

# Line is a sequence of points:

**def** bellen**(**self**)** **:** **pass**

"""

Beim definieren einer class kann man erben in dem man im Parameter den Namen dieser Klasse angibt.

Wird verwendet für attribut referenzen: Wenn ein benötigtes attribut nicht in dieser Klasse gefunden wird,

wird in der Basis Klasse(welche im Parameter übergeben wurde) gesucht.

Abgeleitete klassen können methoden von der Basis Klasse überschreiben

"""

# Verschiedene Strategys

**class** **LautBellen(**BellVerhalten**):**

**def** bellen**(**self**):**

**return** "GANZ LAUT BELLEN!!" # Dummy

**class** **LeiseBellen(**BellVerhalten**):**

**def** bellen**(**self**):**

**return** "ganz leise bellen..." # Dummy

**class** **ElektronischBellen(**BellVerhalten**):**

**def** bellen**(**self**):**

**return** "Elekkkkktronisch Bellen!" # Dummy

**class** **AngstBellen(**BellVerhalten**):**

**def** bellen**(**self**):**

**return** "bellen aus Angst!" # Dummy

# Diese Klasse ist für die Steuerung:

**class** **Hund:**

**def** \_\_init\_\_**(**self**,** strategy**):**

self**.**strategy **=** strategy

**def** bellen**(**self**):**

**return** self**.**strategy**.**bellen**()**

**def** changeBellVerhalten**(**self**,** newBellVerhalten**):**

self**.**strategy **=** newBellVerhalten

# Beispiel-Client

meinHund **=** Hund**(**LautBellen**())**

**print(**meinHund**.**bellen**())**

# Die Strategy ändern

meinHund**.**changeBellVerhalten**(**LeiseBellen**())**

**print(**meinHund**.**bellen**())**

# Quellen

Kompletter Code siehe GitHub

<https://github.com/akrickl-tgm/pattern_python>

*[1] Python Software Foundation; „Documentation >> The Python Tutorial >> 9.5. Ineritance“; Online:* [*https://docs.python.org/2/tutorial/classes.html*](https://docs.python.org/2/tutorial/classes.html)*; zuletzt aufgerufen: 21.12.2014; zuletzt geändert: 12.10.2014  
[2] „Strategy: Choosing the Algorithm at Runtime“; Online:* [*http://python-3-patterns-idioms-test.readthedocs.org/en/latest/FunctionObjects.html#strategy-choosing-the-algorithm-at-runtime*](http://python-3-patterns-idioms-test.readthedocs.org/en/latest/FunctionObjects.html#strategy-choosing-the-algorithm-at-runtime)*; zuletzt aufgerufen: 21.12.2014; zuletzt geändert: 10.12.2014*

*[3] Philipp Hauer; „Das Strategy Design Pattern“; Online:* [*http://www.philipphauer.de/study/se/design-pattern/strategy.php*](http://www.philipphauer.de/study/se/design-pattern/strategy.php)*; zuletzt aufgerufen: 21.12.2014; zuletzt geändert:*

*[4] Wikipedia; „Command Pattern“; Online:* [*http://en.wikipedia.org/wiki/Command\_pattern*](http://en.wikipedia.org/wiki/Command_pattern)*; zuletzt aufgerufen: 21.12.2014; zuletzt geändert: 15.12.2014*

*[5] Philip Hauer; „Das Command Design Pattern“; Online:* [*http://www.philipphauer.de/study/se/design-pattern/command.php*](http://www.philipphauer.de/study/se/design-pattern/command.php)*; zuletzt aufgerufen: 21.12.2014*

*[6] „Command: Choosing the Operation at Runtime“; Online:* [*http://python-3-patterns-idioms-test.readthedocs.org/en/latest/FunctionObjects.html#command-choosing-the-operation-at-runtime*](http://python-3-patterns-idioms-test.readthedocs.org/en/latest/FunctionObjects.html#command-choosing-the-operation-at-runtime)*; zuletzt aufgerufen: 21.12.2014; zuletzt geändert: 10.12.2014*