Marlene Dorfinger, Christian Maran

Design-Patterns in Python

Inhalt

[1. Aufgabenstellung 2](#_Toc407130411)

[2. Zeitaufzeichnung 2](#_Toc407130412)

[3. Patterns 2](#_Toc407130413)

[Singleton Pattern 2](#_Toc407130414)

[Observer pattern 3](#_Toc407130415)

[Decorator Pattern 5](#_Toc407130416)

[Factory Pattern 5](#_Toc407130417)

[Adapter Pattern 5](#_Toc407130418)

[Quellen 5](#_Toc407130419)

# 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  
http://www.vincehuston.org/dp/

head first: design pattern

# Zeitaufzeichnung

|  |  |  |  |
| --- | --- | --- | --- |
| Aufgabe | Erwartete Zeit | Tatsächliche Zeit | Zuständigkeit |
| Singleton Pattern | 30 min | 20 min | Dorfinger |
| Observer Pattern | 30 min | 30 min | Dorfinger |
| Decorator Pattern | 20 min | 15 min | Maran |
| Factory Pattern | 20 min | 17 min | Maran |
| Adapter Pattern | 20 min | 15 min | Maran |
| Protokoll | 40 min | 30 min | Dorfinger |
| Gesamt | 2 h 40 min |  |  |

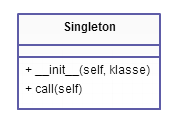
# Patterns

## Singleton Pattern

Beschreibung:

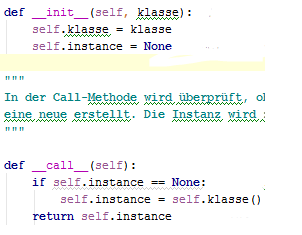
Das Singleton Pattern stellt sicher, dass nur genau eine Instanz einer Klasse erzeugt wird.

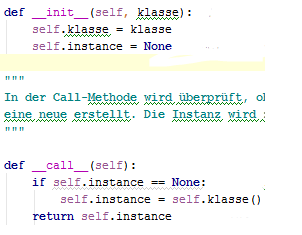
UML:

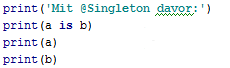
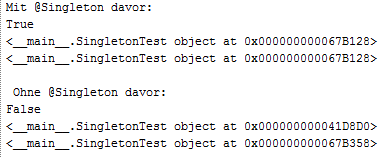
**

Umsetzung:

Um das Singleton-Pattern in Python umzusetzen, braucht man 2 Methoden. \_\_init\_\_ setzt die Klasse, von der es nur eine Instanz geben soll und setzt die Instanz auf none.   
In der \_\_call\_\_ Methode wird überprüft, ob die Instanz None ist, wenn ja dann wird eine neue Instanz der Klasse erstellt und zurückgegeben. Wenn nicht dann wird die alte Instanz zurückgegeben.





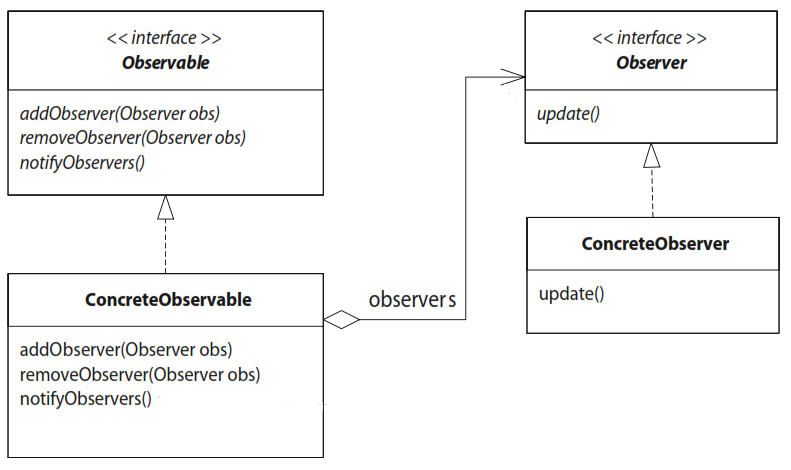
Um das Singleton-Pattern nun anzuwenden, muss man die Klasse mit einem @Singleton vor dem class nameDerKlasse kennzeichnen. Als Klasse wird die Klasse, die danach mit class kommt, gesetzt.   
  
Wenn man nun diese Klasse 2 Mal neu erstellt und entweder „a is b“ oder a und b (also die Speicherstellen) ausgibt, kann man sehen, dass es gleich ist (True bzw gleiche Speicherstelle). Wenn man das @Singleton von der Klasse wegtut und die ganzen Schritte wiederholt, sieht man, dass nun „False“ bzw eine unterschiedliche Speicherstelle rauskommt.   
  
  
Ausgabe:  


## Observer pattern

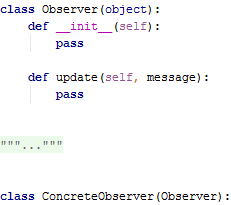
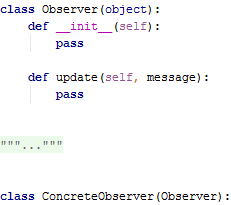
Beschreibung:

Das Observer Pattern ermöglicht, dass sich Objekte (Observer) bei einem anderem Objekt (Observable) registrieren und von diesem informiert werden (können auch wieder gelöscht werden).

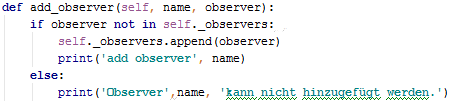
UML:

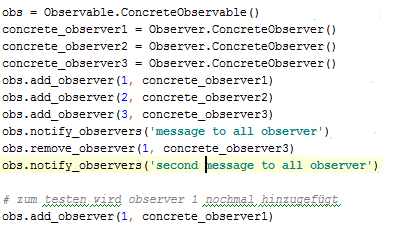


Umsetzung:

Das Problem an der Umsetzung ist, dass es in Python keine Interfaces wie in Java gibt. Das löse ich mit Vererbung. Ich schreibe eine Klasse nur mit den Methodenköpfen und in den Methoden nur pass. Diese Art hab ich im Internet gefunden.   
Dann schreibe ich in den vererbbaren Klassen als Parameter bei class-Zeile die Klasse die man erben möchte. Bei mir sieht das so aus:  
  


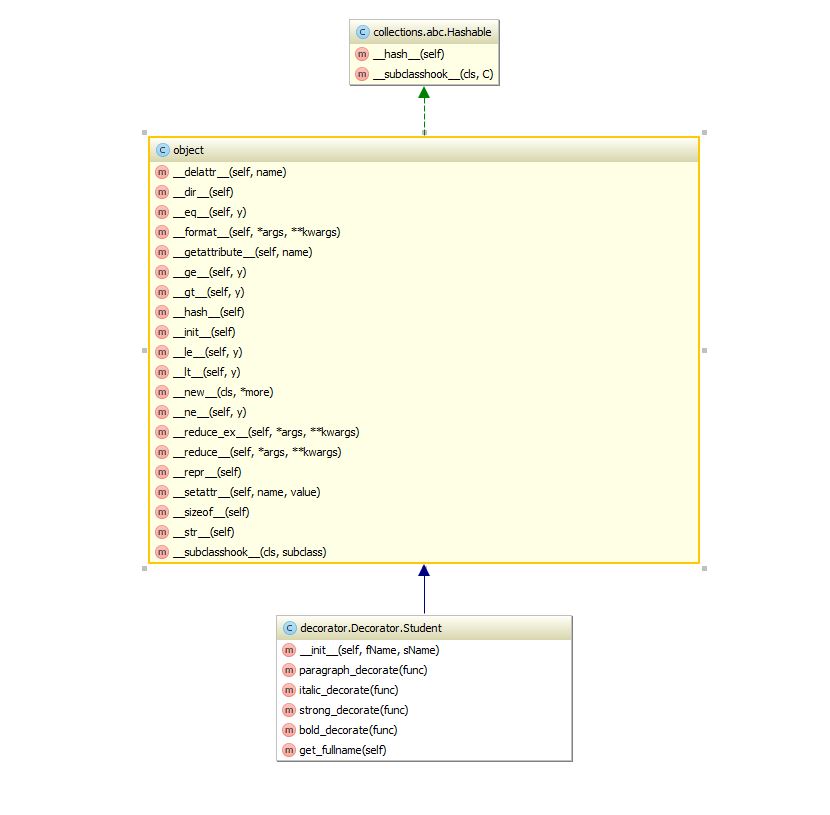
Damit die Vererbung funktioniert, habe ich die Super-Klasse und die Klasse die erbt in ein Python-File gegeben.

Die restliche Umsetzung ist dann einfach, man muss die ganzen Methoden ausprogrammieren. In Observable hat man eine Liste (in Python muss man die Länge der Liste nicht angeben) und in diese fügt man ein Element hinzu oder löscht es.   


Um Die Observer zu benachrichtigen braucht man eine notify\_observers Methode, diese geht alle Elemente der observers-Liste mit einer for-Schleife durch, dann führt man die update-Methode des Observers aus. Das ist in meinem Fall, dass die Nachricht die mitgegeben wird, ausgegeben wird.   
  
Um zu sehen ob es funktioniert, habe ich eine eigene Klasse TestObserver erstellt. Diese erstellt ein neues ConcreteObservable und 3 ConcreteObservers. Ich habe dann alle 3 Observer dem Observable hinzugefügt und dann die notify\_observers-Methode aufgerufen. Die mitgegebene Nachricht wurde dann 3 Mal ausgegeben (weil 3 Observer). Anschließend habe ich einen Observer gelöscht und nochmal die notify-Methode ausgeführt, die Nachricht wurde nur 2 Mal ausgegeben (weil nur mehr 2 Observer). Um die add-Methode zu testen, hab ich dann einen schon vorhandenen Observer nochmal hinzugefügt. Das wurde durch meine Fehlermeldung vermieden.   


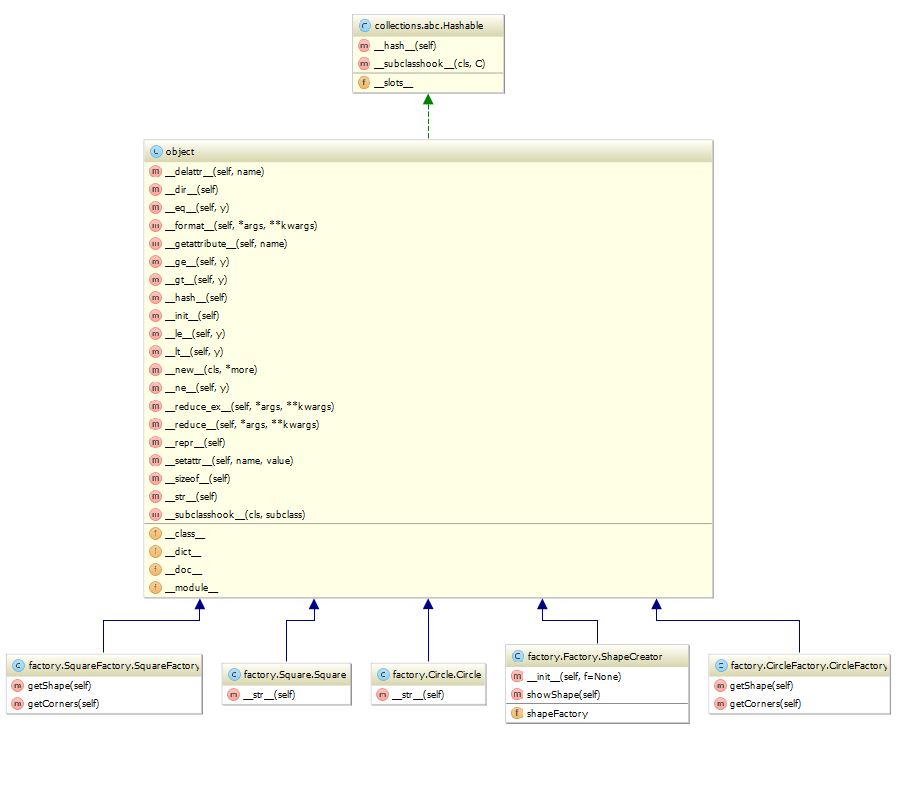
## Decorator Pattern

Python ist reich mit leistungsstarken Funktionen. Wrapper(decorator) sind welche davon. Im Rahmen der Design-Patterns, wollen wir dynamisch die Funktionalität einer Funktion, Methode oder Klasse ändern, ohne direkt am Code des Objekts was zu ändern.

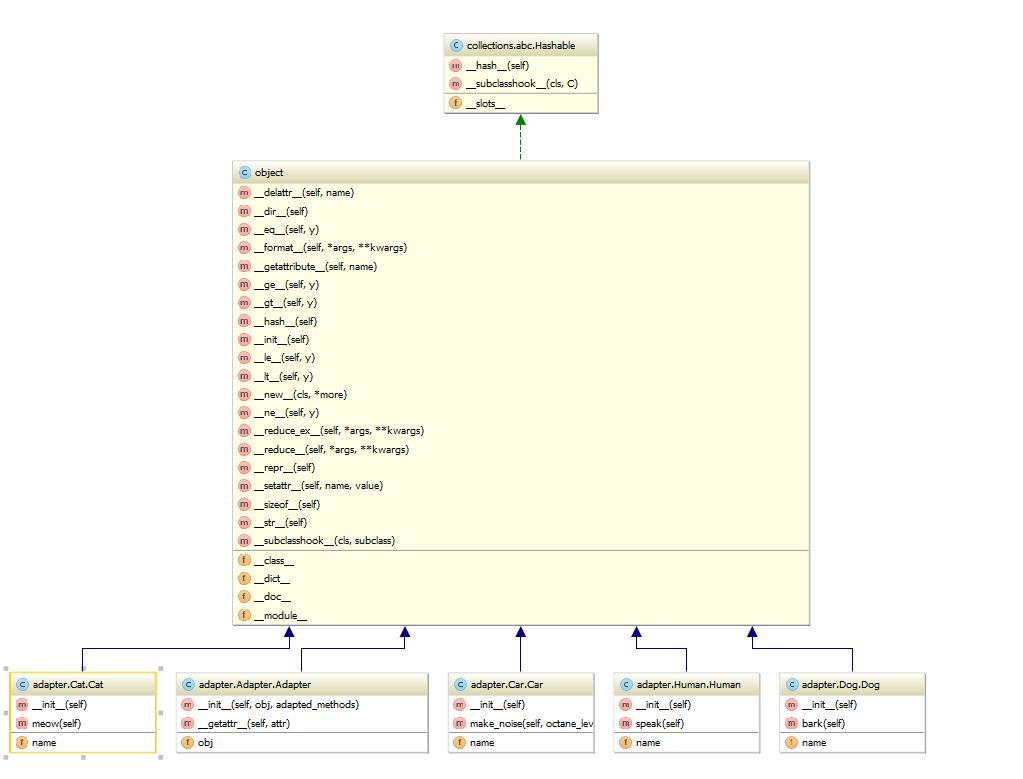


## Factory Pattern

Wenn Sie feststellen, dass Sie neue Arten von „Objekten“ zu einem System hinzufügen müssen, ist es das Sinnvollste, Polymorphismus zu verwenden, um eine gemeinsame Schnittstelle zu den neuen Typen zu erstellen. Dies trennt den Rest des Codes in dem System von der Kenntnis der spezifischen Typen, die Sie hinzufügen. Neue Arten können ohne störenden vorhandenen Code hinzugefügt werden ... so scheint es. Zunächst scheint es, dass der einzige Ort, um den Code in einem solchen Design zu ändern, der Punkt ist wo ein neuer Typ übernommen wird, aber das ist nicht so ganz richtig. Man muss noch immer ein Objekt vom neuen Typ erstellen und genau angeben welcher Konstruktor verwendet wird



## Adapter Pattern



# Quellen

<http://stackoverflow.com/questions/42558/python-and-the-singleton-pattern>   
gesehen 22.12.2014

<https://razvantudorica.com/08/example-for-singleton-decorator-pattern-in-python/>   
gesehen 22.12.2014

<https://github.com/maigfrga/blog/blob/master/2013/09/observer-pattern/python/observer.py>   
gesehen 23.12.2014

Patterns Kompakt 4. Auflage – Karl Eilebrecht, Gernot Starke