1 Objektorientiertes Programmieren – OOP

Das folgende Skript ist als kurzer und schneller Überblick über die Objektorientierung zu verstehen. Nach der Lektüre werden viele Fragen unbeantwortet bleiben und wahrscheinlich noch mehr Fragen zusätzlich aufgetaucht sein.

Eine vollständige ausführliche Abhandlung über dieses Thema wäre eine eigene Vorlesung wert. Wer tiefer in die Materie eintauchen will (oder dieses Skript ganz einfach fürchterlich findet), dem seien folgende Webseiten und Bücher empfohlen:

- **http://openbook.galileocomputing.de/python/** – Galileo Computing Open Book über Python mit einer guten Einführung in das objektorientierte Programmieren in Python.
- **http://openbook.galileocomputing.de/oo/** – Galileo Computing Open Book zum Thema Objektorientierung im Allgemeinen. Beide Open Books liegen als ZIP-Dateien vor, so dass man die Bücher auch offline lesen kann.
- **Effective C++** und
- **More Effective C++** – beide von Scott Meyers. Beide Bücher behandeln zwar C++, jedoch sind die Konzepte und Ideen, die besprochen werden, für die meisten objektorientierten Sprachen übertragbar und die Beispiele sind auch für C++-Laien verständlich.

1.1 Motivation

Warum ist OOP so toll? Worin besteht der Vorteil darin objektorientiert zu programmieren, wenn man doch alles auch funktional oder prozedual programmieren kann? Und was kann man mit OOP machen, was man sonst nicht kann?

Hier sind einige Antworten auf diese Fragen:

- OOP ermöglicht das Management großer Projekte.
- OOP erleichtert das verteilte Programmieren.
- OOProgramme sind besser wartbar.
- OOP erhöht die Lesbarkeit von Programmen und deren Logik.
- OOP unterstützt bei der Wiederverwertung bereits geschriebenen Codes.
- OOP-Code ist besser testbar.
- OOP eignet sich besser zur Erweiterung von vorhandenen Programmen.

Na, wenn das keine Anreize sind ...

1.2 Konstrukte der OOP

In der OOP greift man auf drei Programmierkonstrukte zurück:

Klassen Eine Klasse repräsentiert quasi die Idee eines Objekts. In der Klasse definiert man die Eigenschaften und Aktionsmöglichkeiten der später davon abgeleiteten Objekte.

Objekte Objekte sind Instanzen (Repräsentanten) einer Klasse. Bei der Erzeugung der Objekte wird ihnen von der Klasse ihre unveränderlichen Eigenschaften und Interaktionsmöglichkeiten mit auf den Lebensweg gegeben.

Member Datenmember oder Funktionsmember werden häufig auch Instanzvariablen bzw. Methoden genannt. Dies sind die Eigenschaften (Datenmember) und die Interaktionsmöglichkeiten (Funktionsmember) der Objekte.

In der OOP haben sich viele verschiedene Ausdrücke für die gleichen Dinge eingebürgert. So bezeichnet man z.B. eine abgeleitete Klasse auch häufig als Kindklasse, Unterklasse, oder als Subklasse und übergeordnete Klassen als Elternklasse, Oberklasse, oder als Superklasse – gemeint ist aber jeweils das selbe.

1.3 Die drei Grundelemente der Objektorientierung

Das Thema objektorientierte Programmierung kann man auf drei grundlegende Konzepte vereinfachen:

1. Polymorphismus
2. Datenkapselung
3. Vererbung

Programmiersprachen, die sich objektorientiert nennen wollen, müssen sich daran messen lassen, wie stark sie diese Konzepte verwirklichen (im höchsten Grade tut dies die Sprache Smalltalk). Wir werden sehen, dass Python z. B. das Thema Datenkapselung nicht so implementiert, wie sich das der Vater der Objektorientierung Alan Kay gedacht hat.

1.3.1 Polymorphismus

Unter Polymorphismus versteht man die Möglichkeit, ein und denselben Methodennamen mit unterschiedlichen Funktionalitäten zu belegen. Zum Beispiel sagt man umgangssprachlich, dass sowohl eine Rakete, als auch ein Helikopter »starten«und danach in der Luft sind, obwohl der Startvorgang bei beiden Fluggeräten sich sehr unterschiedlich darstellt. Trotzdem benutz man denselben Methodennamen.

In der OOP kann man nun mittels der Polymorphie zwei Klassen definieren (die nicht einmal miteinander verwandt sein müssen), die beide eine Methode `xyz` implementieren. Der Interpreter/Compiler wählt zur Laufzeit des Programms die passende Version der Methode je nachdem, auf welches Objekt welcher Klasse die Methode angewandt wird.

1.3.2 Datenkapselung

Man stelle sich vor, man programmiert eine Klasse mit einer Menge an Instanzvariablen und Methoden mit einer unglaublich ausgeklügelten Funktionalität. Diese Klasse gibt man an seinen nicht ganz so talentierten Kollegen weiter, der die Idee der Klasse wohl nicht ganz so verinnerlicht hat. Das heißt, er benutzt sie nicht wie vorgesehen und übergibt den Objekten falsche bzw. unsinnige Werte für Instanzvariablen, oder er ruft Methoden auf, die eigentlich nur als Hilfsmethoden für andere Methoden gedacht sind, usw.. Das wäre jetzt eigentlich nicht so schlimm, würde der Kollege nur für sich selbst programmieren. Tut er dies jedoch für andere, oder innerhalb eines großen Projekts, so kann sich daraus schnell ein Problem ergeben.

Man muss die Klasse also vor solch unbedarften Gebrauch schützen, indem man seine Daten und Methoden vor der Außenwelt versteckt bzw. abkapselt.

Dazu ordnet man die Methoden und Daten in eine von drei Zugriffsklassen ein:

Public bzw. Öffentliche Member (Daten und Methoden) sind von außen sowohl les- als auch schreibbar.

Protected bzw. Geschützte Member können in der Klasse, in der sie deklariert wurden gelesen und geschrieben werden, aber auch in allen davon abgeleiteten Klassen.

Private Member sind lediglich in der Klasse, in der sie definiert wurden, zugänglich für Lese- und Schreibzugriff.

In Python existieren zwar Möglichkeiten, die Member in eine der drei Zugriffsklassen einzuordnen – es bestehen jedoch trotzdem Möglichkeiten, die Einschränkungen der Zugriffsklassen zu umgehen. Python gibt also sozusagen lediglich »Empfehlungen«, wie bestimmte Member zu behandeln sind. Diese Empfehlungen werden vom Programmierer sicher nicht zum Spaß ausgesprochen – also halten wir uns auch dran!

1.3.3 Vererbung

Das Konzept der Vererbung ist sicherlich das interessanteste in der OOP. Es besagt, dass Klassen ihre Methoden und Daten an abgeleitete Klassen vererben. Man muss also nicht jedes mal alles wieder neu schreiben in einer Klasse `Lkw`, wenn man die Klasse `Kfz` schon implementiert hat. Man hat ja geerbt.

Das heißt aber nicht, dass man nichts Neues schreiben darf. Zusätzliche Methoden und Daten sind dann aber nur in der abgeleiteten Klasse bekannt, nicht darüber. Außerdem besteht die Möglichkeit geerbte Methoden neu zu implementieren und sie somit an die neue Klasse anzupassen. Der Zugriff auf die entsprechende Methoden der übergeordneten Klassen besteht dann jedoch weiterhin (in Python geschieht dies durch die Nennung der Superklasse vor dem Methodennamen getrennt durch einen Punkt `Superklasse.Methodenname`).

1.4 Syntax der OOP in Python

Das folgende Listing steht exemplarisch für die Syntax der Klassendefinition in Python:

```
1  class Klassenname(Oberklasse):  # Definition der Klasse
2          Klassenvariable         # Klassenvariable bzw. statische Variable
3      def __init__(self, listeOptionalerParameter):   # Konstruktor
4          # Aufruf des Konstruktors der Oberklasse
5          Oberklasse.__init__(self, parameterDerOberklasse)
6          # Initialisierung der Attribute (private)
7          self.__datenMember1 = irgendeinWert
8          # Initialisierung der Attribute (protected)
9          self._datenmember2 = irgendeinAndererWert
10         # Initialisierung der Attribute (public)
11         self.datenmember3 = irgendeinNochAndererWert
12         # ...
13         # ...
14
15     def __del__(self):  # Destruktor
16         del self.__datenMember1     # Loeschen der Attribute
17         del self._datenMember2
18         del self.datenMember3
19         Oberklasse.__del__(self)    # Aufruf des Destruktors der Oberklasse
20
21     def methode1(self, listeOptionalerParameter):   # Methodendefinition (public)
22         # logischer Code
23         # ...
24         # ...
25         # ...
26         return etwas    # Rueckgabewert (optional, aber waermstens empfohlen)
27
28     def _methode2(self, listeOptionalerParameter):   # Methodendefinition (protected)
29         # logischer Code
30         # ...
31         # ...
32         # ...
33         return etwas    # Rueckgabewert (optional, aber waermstens empfohlen)
34
35     def __methode3(self, listeOptionalerParameter):   # Methodendefinition (private)
36         # logischer Code
37         # ...
38         # ...
39         # ...
40         return etwas    # Rueckgabewert (optional, aber waermstens empfohlen)
41
42     def klassenMethode(listeOptionalerParameter):   # Definition einer Klassenmethode
43         # logischer Code
44         # ...
45         # ...
46         # ...
47         return etwas    # Rueckgabewert (optional, aber waermstens empfohlen)
48
49     klassenMethode = staticmethod(klassenMethode)
```

- In Zeile 1 steht die Definition der Klasse mitsamt der Angabe, von welcher Oberklasse abgeleitet wird. Alles weitere muss ab jetzt eingerückt werden.
- In der Zeile 2 wird eine Klassenvariable (auch statische Variable genannt) definiert.
- In Zeile 3-13 wird der Konstruktor der Klasse implementiert.

- In Zeile 5 sieht man, wie eine Methode einer übergeordneten Klasse aufgerufen wird.
- In den Zeilen 7-11 werden die Datenmember mit den Zugriffsrechten private, protected und public initialisiert.
- Der Destruktor wird in den Zeilen 15-19 implementiert.
- Die Methoden der Klasse beginnen ab der Zeile 21 (public) bzw. ab der Zeile 28 (protected) und ab der Zeile 35 (private).
- In den Zeilen 42-47 wird eine Klassenmethode bzw. statische Methode implementiert. Dazu muss diese aber noch als solche angemeldet werden. Dies geschieht in Zeile 49.