3 Vererbung und Klassendiagramme

3.1 Objektorientierte Modellierung und UML

In der objektorientierten Programmierung haben Klassen eine Vielzahl von Aufgaben. Das primäre Ziel jedoch ist die Gruppierung respektive Kapselung von Attributen und dazugehörigen Methoden, vor allem aus dem Grund der Wiederverwendbarkeit. Die Gruppierung entspricht einer konzeptuellen Einheit, welche durch den Klassennamen repräsentiert wird.

Andere wichtige Gründe für die Anwendung der Objektorientierung sind die Verkürzung der Entwicklungszeit, die Senkung der Fehlerrate und eine verbesserte Erweiterbarkeit und Anpassungsfähigkeit.

Ein wichtiger Aspekt der Objektorientierung ist die **objektorientierte Modellierung**. Bei der objektorientierten Modellierung geht es darum, Anforderungen zu erfassen und zu beschreiben, welches das endgültige Produkt haben soll. Dies geschieht zum Beispiel in der Form eines Pflichtenheftes.
Sind die Anforderungen bekannt, so kann das werdende Softwareprodukt in Form eines Diagrammes beschrieben werden. Wir unterscheiden zwei wichtige Typen von Modellen:

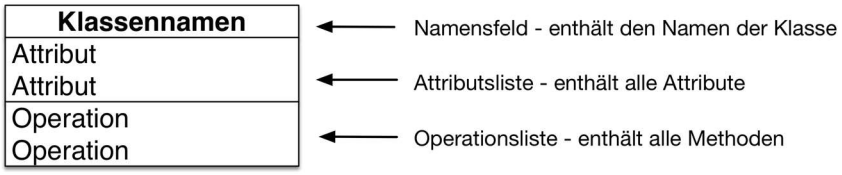
- Dynamische Modelle, z.B. Flussdiagramme, Aktivitätsdiagramme
- Statische Modelle, z.B. Klassendiagramme

Es gibt Regeln, wie solche Diagramme erstellt werden. Die **Unified Modelling Language** (kurz UML) ist die grafische Modellierungssprache, welche am häufigsten benutzt wird um Software-Projekte zu modellieren/realisieren. UML besteht aus Sprache, Beziehungen, grafischer Notation und Austauschformat. Wir werden vor allem die grafische Notation betrachet.

Es existieren verschiedene Typen von Diagrammen, auf die wir nicht alle detailliert eingehen können. Wir beschränken uns hier auf das "**UML Klassendiagramm**". Weitere wichtige Diagramme von UML sind das Objektdiagramm, Aktivitätäsdiagramm, Sequenzdiagramm und das Anwendungsfalldiagramm.

3.2 Darstellung von Klassen in UML

Das UML Klassendiagramm dient dazu, Klassen **unabhängig von der Programmiersprache abzubilden**. Es können die Klassen selbst und deren Beziehungen zu anderen Klassen dargestellt werden.



Die Darstellung einer Klasse in UML ist einfach. Es wird ein Rechteck genommen und darin steht der Klassennamen. Besitzt die Klasse Attribute und Methoden, so werden diese durch weitere Rechtecke erweitert, wie in der Abbildung oben.

UML Diagramme werden oft mit Softwaretools wie Visio (Windows) oder OmniGraffle (Mac) erstellt. Es gibt auch komplexe Softwarepakete, welche auf UML spezialisiert sind, welche allerdings für kleinere bis mittlere Projekt oft ungeeignet sind (z.B. Borland Together, ArgoUML, ...) Ein sehr "einfaches" Vektorprogramm, oder noch besser - eine Skizze von Hand ist als Entwurf oft besser geeignet! In Visual Studio Code gibt es auch die Extension PlantUML.

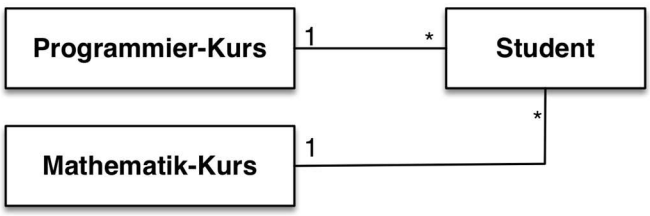
3.3 Objektorientierte Beziehungen

Nachdem wir in der Lage sind Klassen in der UML Notation darzustellen, ist es wichtig auch die Beziehungen zwischen Klassen anzusehen. Es gibt dabei verschiedene Beziehungstypen zwischen Klassen.

3.3.1 Assoziation ("hat-Beziehung")

Eine Assoziation beschreibt eine Beziehung zwischen zwei (oder mehr) Klassen. Ein Objekt besitzt ein oder mehrere Objekte. Bei der Assoziation haben alle Objekte ihren eigenen Lebenszyklus und werden separat erstellt. Die Klassen haben keinen Besitzer. Die assoziierten Objekte können von unterschiedlichen Klassen benutzt werden.

UML Notation: Strich und Angabe der Multiplizität. Ein (offener) Pfeil ist optional.



Beispiel: Assoziation Student und Kurs. "Kurs hat Student"

In Python wird die Assoziation oft so realisiert, dass die Objekte ausserhalb erstellt werden und über eine Methode zugewiesen werden. Hier eine mögliche Implementierung der oben gezeigten Klassendefinition:

```
class Student:
   def __init__(self, ...):
      self.vorname = ...
      self.name = ...

class ProgrammierKurs:
   def __init__(self):
      ...
   def AddStudent(student):
```

```
      ...

class MathematikKurs:
   def __init__(self):
      ...
   def AddStudent(student):
      ...
```

Der Lebenszyklus der Instanzen wird also "von aussen" definiert:

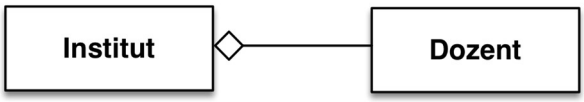
```
student = Student("Martin", "Müller)
kurs1 = ProgrammierKurs()
kurs2 = MathematikKurs()

kurs1.AddStudent(student)
kurs2.AddStudent(student)
```

3.3.2 Aggregation ("besteht-aus-Beziehung")

Die Aggregation ist eine spezielle Form der Assoziation. Ein Objekt besteht aus einem (oder mehreren) anderen Objekten. Die Objekte haben auch ihren eigenen Lebenszyklus, aber es gibt genau einen Besitzer eines Objektes.
Die einzelnen Komponenten des Aggregators sind auch ohne andere Klasse existierend zu benutzen.

UML-Notation: Raute und Strich und Angabe der Multiplizität.

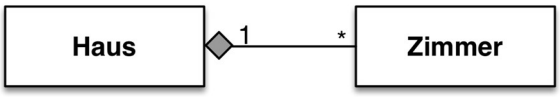


Beispiel: Institut und Dozent: Der Dozent gehört in diesem Beispiel genau einem Institut an. Es wäre aber möglich das Institut zu wechseln.

3.3.3 Komposition ("starke besteht-aus-Beziehung")

Ein Objekt besteht aus einem (oder mehrerer) anderen Objekten. Die einzelnen Komponenten des Aggregators sind ohne die andere Klasse nicht existent und nicht zu benutzen. Die beiden Klassen haben einen gemeinsamen Lebenszyklus und können nicht alleine existieren.

UML-Notation: ausgefüllte Raute und Strich und Angabe der Multiplizität.



Beispiel: Haus und Zimmer: Das Zimmer kann das Haus nie wechseln!

Mögliche Implementierung:

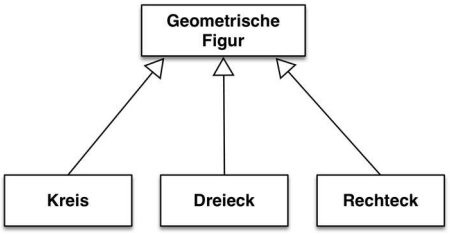
```
class Zimmer:
   def __init__(self):
      ...

class Haus:
   def __init__(self):
      self.Zimmer1 = Zimmer()
      self.Zimmer2 = Zimmer()
```

3.3.4 Vererbung ("ist-Beziehung")

Die Vererbung ist eine Beziehung beider alle Attribute und Methoden der Elternklasse übernommen wird und ergänzt werden kann.

UML Notation: Pfeil mit unausgefüllter Pfeilspitze ("Dreieck") auf übergeordnete Klasse.



Beispiel: Geometrische Figur vererbt Kreis, Dreieck, Rechteck

3.3.4.1 Beispiel Code für Vererbung: Figur

```
class Figur:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def flaeche(self):
        return 0
```

figur.py

```
from figur import *
```

```python
class Kreis(Figur):
    def __init__(self, mittelpunkt, radius):
        super().__init__(self, "Kreis")
        self.mittelpunkt = mittelpunkt
        self.radius = radius

    def flaeche(self):
        return self.radius**2 * math.pi
```

kreis.py

3.3.4.2 Beispiel für Vererbung: Fahrzeug

Wir schreiben eine Simulation. Fahrzeuge sind:

Fahrrad, PKW, LKW, Lieferwagen, Polizeiauto.

Fahrzeuge haben:

- Anzahl Räder
- Klassenspezifische Attribute / Methoden
- Eine maximale Geschwindigkeit (`getMaxVelocity()`)
  Eine Methode `move()`, welche aufgrund der Maximalgeschwindigkeit sich innerhalb einer Minute x Meter weit bewegt (Zufallsgenerator)
  [km/h / 3.6 -> m/s]

Verschiedene Fahrzeuge sollen in einer Klasse “Flotte” verwaltet werden und in einem Wettrennen bewegt. Sieger ist, wer zuerst 100km zürückgelegt hat.