UML

Modelle: Modelle helfen uns Systeme zu visualisieren, die Struktur oder das Verhalten eines Systemes festzulegen und liefern uns eine Schablone die uns beim Konstruieren des Systems hilft. Außerdem dokumentieren Modelle getroffene Entscheidungen. Es gibt dabei für jeden Zweck ein eigenes Modell.

UML Werkzeuge: Einige Typische Funktionalitäten die UML Werkzeuge liefern sind:

- Diagrammunterstützung: Erzeugen und bearbeiten von UML-Diagrammen
- Quelltexterzeugung: Code wird aus UML-Diagramm erzeugt
- Reverse Engineering: Aus Code wird ein UML-Diagramm erzeugt
- „Roundtrip"-Engineering: Code und Diagramme werden ständig in Sync gehalten

Stufen der OOP-Softwareentwicklung:

- Objektorientierte Analyse (OOA) - unterstützt durch UML. Die Aufgabenstellung wird gründlich erfasst und analysiert. Es werden Klassen konzipiert und Beziehungen zwischen diesen festegelegt.
- Objektorientiertes Design (OOD) - unterstützt durch UML. Die gefunden Klassen werden nun an die technische Platform angepasst und eine Benutzeroberfleche wird entwickelt. Es wird sich damit beschäftigt wie das lauffähige Programm selbst organisert sein soll.
- Objektorientierte Programmierung (OOP)

Anwendungsfalldiagramm

Die Ergebnisse einer Anwendungsfallmodellierung sollen sein:

- Mehrere Anwendungsfalldiagramme.
- Eine detaillierte textuelle Beschreibung für jeden Anwendungsfall.
- Eine kurze Beschreibung zur Abgrenzung: Was soll das System **nicht** leisten.

System: Was wird beschrieben?

Akteure: Wer benutzt das System? (Sind aktive Teilnehme die Prozesse in Gang setzen oder Prozesse am Laufen halten.)

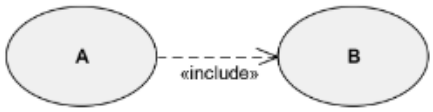
Use Cases: Was machen die Akteure? (Dokumentieren die typischen Prozeduren aus Sicht der Akteure für ausgewählte Fälle.). Diese müssen immer einen Nutzen für den Akteur darstellen. Werden als Ellipse gezeichnet.

Assoziationen: Ein jeder Akteur ist mit eine Assoziation mit einem Use Case Verbunden, was heißt er führt die Prozedur durch. Dabei kann es Multiplizitäten auf seiten der Akteure geben, was nichts anderes heißt als, dass ein Use Case nur von x Akteuren durchgeführt werden kann.

Beziehungen: Use Cases können mittels einer gestrichelten Linie mit Pfeil auch mit anderen verbunden werden. Dabei gibt es mehrere Arten:

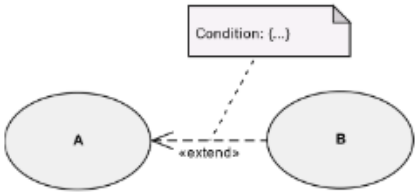
- <<include>>: Bedeutet, dass der Use-Case mit der Pfeilspitze ausgeführt werden muss um den eigentlichen Use-Case auszuführen. Also quasi eine Benötigt Beziehung.
- <<extend>>: Bedeutet, dass der Use-Case mit der Pfeilspitze entscheidet ob der andere Use-Case ausgeführt wird - muss aber nicht. Man kann dabei immer Expensionpoints angeben, welche aussagen wann etwas eingefügt wird. Man gibt diese mit einer Notiz mit dem Text "Condition: {...}" an. Außerdem schreibt man in der Ellipse mit der Pfeilspitze eine Linie nach dem Namen und dann den Namen des Extensionpoints (dieser kann in der Notiz angegeben werden, oder ist gleich des Namens des Use-Cases das extended wird.)

- **<<include>>** entspricht Unterprogrammaufruf.



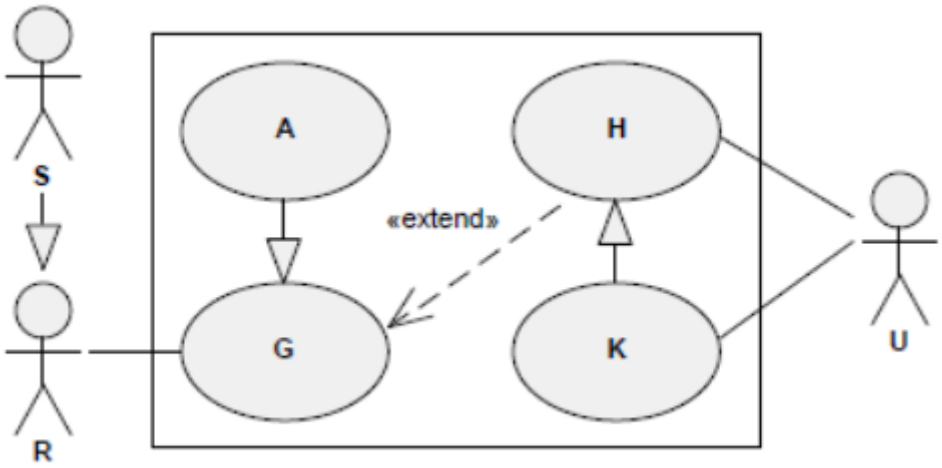

```
def B():
    …
def A():
    …
    B()
    …
```

- **<<extend>>** entspricht bedingtem Unterprogrammaufruf.




```
def B():
    …
def A():
    …
    if Condition: B()
    …
```

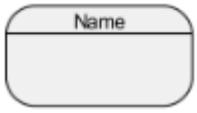
Generalisierung: Ein Offener Pfeil bedeutet: "ist eine Generalisierung von". Dies kann sowohl bei Use-Cases als auch bei Akteuren verwendet werden. Wird es bei UseCases genutzt, dann muss man aufpassen, da alle Assoziationen der Oberklasse geerbt werden! Dies heißt auch dass so wie hier dann zwei Us benötigt werden:



Zustandsdiagramm

Zustand:

- echter Zustand: System kann sich dauerhaft im Zustand befinden. Also ein Zustand im eigentlichen Sinne

  oder ein Endzustand:

- Pseudozustand: System kann nicht dauerhaft in einem Pseudozustand sein. Also z.B. ein Startzustand

  aber auch ein Entscheidungsknoten usw.

Aktivitäten innerhalb eines Zustands:

- entry / aktivität: Wird beim Eingang in den Zustand ausgeführt
- exit / aktivität: Wird beim Verlassen des Zustands ausgeführt
- do / aktivität: Wird kontinuierlich ausgeführt.
- event / aktivität: Wird ausgeführt wenn ein Event eintritt.

Zustandsübergänge: Erfolgt wenn ein Ereignis eintritt und die Bedingung erfüllt ist. Bei nicht Erfüllung geht das konsumierte Ereignis verloren!

Default Werte für Zustandsübergänge: Fehlendes Ereignis entspricht dem Ereignis "Aktivität ist abgeschlossen". Fehlende Bedingung entspricht true.

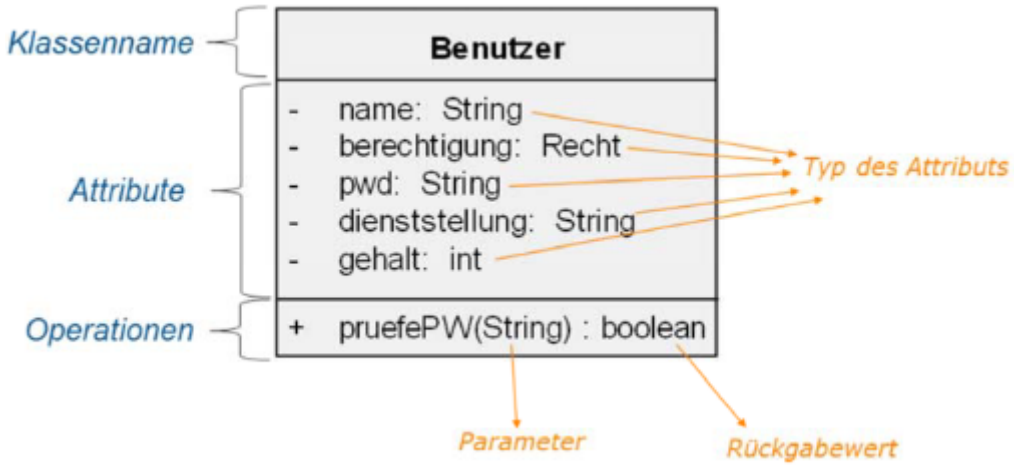
Zustandsübergänge - Syntax: Notation: `Ereignis (Argumente) [Bedinungen] / Aktivität`

Zustandsübergang - Ereignistypen:

- CallEvent: Empfang einer Nachricht
- SignalEvent: Empfang eines Signals
- ChangeEvent: Eine Bedingung wird wahr - also Bedingung wird permanent geprüft wohingegen eine Bedingung alleine nur geprüft wird wenn ein zugeordnetes Ereignis eintritt. Syntax: ChangeEvent: `when(B)`
- TimeEvent: Zeitablauf oder Zeitpunkt

Klassendiagramm

Klasse: Dargestellt als Rechteck. Besteht aus einem Namen, Attribute und Operationen.



Attribute und Operationen:
Sichtbarkeit:

- + ... public
- - .... private
- # .... protected
- ~ .... package: Zugriff durch Objekte deren Klasse sich im selben Paket befinden, erlaubt
- unterstrichen: Klassenattribute/Operationen
  Eigenschaften: Bei Attributen kann man nach dem default Wert noch Eigenschaften angeben `- name: str = "" {Eigenschaft, Eigenschaft, ...}`.
- readOnly: eine Konstante in Python
- unique: Multi-valued property has no duplicate Values
- ...

Objektbeziehungen:

- *Vererbung*: Dargestellt mit einem offenen Dreieck. Beschreibt eine Generalisierung. Also die Klasse ohne Pfeilspitze "ist eine" Klasse mit Pfeilspitze. Man unterscheidet und schreibt die jeweilige Eigenschaft in geschweiften Klammern neben den Strich:
  - incomplete oder complete: Bei complete kann man die Superklasse alleine nicht initialisieren. Also eine Person ist weder Angestellter noch Student.
  - overlapping oder disjoint: Bei overlapping kann ein Objekt Instanz von mehr als einer Subklasse sein. Also eine Person ist Angestellter UND Student.
- *Assoziation*: Einfache Linie. Beschreibt eine einfache Beziehung. Welche Beziehung, kann als Name angegeben werden.
- *Gerichtete Assoziation*: Einfacher Pfeil mit Kreuz an am anderen Ende oder einfach zwei Pfeile. Dabei heißt dass die Klasse mit der Pfeilspitze nicht zu Objekten der Klasse mit dem X navigieren kann. Umgekehrt allerdings schon.
- *Aggregation*: Linie mit offener Raute am Ende. Beschreibt, dass die Klasse mit der Raute (=das Ganze) aus Teilen der Klasse ohne Raute besteht. Also eine "ist teil" Beziehung.

Wichtig: Die Teile können dabei auch ohne dem Ganzen bestehen.

- *Komposition*: Linie mit geschlossener Raute am Ende. Ist wie die Aggregation, jedoch können Objekte der Klasse ohne Raute nicht ohne einem Objekte der Klasse mit Raute existieren. Insbesondere kann also ein Teil immer nur in einem Ganzen eingebaut sein!

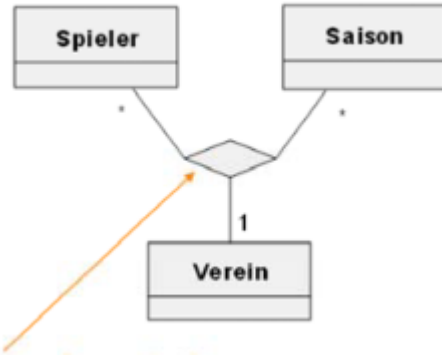
Multiplizitäten: Wird durch einen Regulären Ausdruck oberhalb der Beziehungslinie je neben einer Klasse dargestellt. Dabei ist die Zahl auf der Seite der Klasse B für die Klasse A gültig. Also ein Objekt der Klasse A steht mit Zahl auf Seite B vielen Objekten der Klasse B in Verbindung. Beispiel:

```
A 0...1 -------> 2 B
```

Ein Objekt der Klasse A steht mit genau 2 Objekten der Klasse B in Beziehung. Ein Objekt der Klasse B steht mit höchstens einem Objekt der Klasse A in Beziehung.

WICHTIG: Bei einer Komposition, kann auf der Seite der Raute immer nur eine 1 oder 0 stehen.

N-äre Beziehung: Dargestellt durch Raute mit Verbindungen zu den Klassen. Eine solche Beziehung wird nur genutzt um Einschränkungen auf Multiplizitäten anzugeben. Sind die Klassen A, B und C an einer solchen Beziehung beteiligt, setzt man immer die Zahlen bei A und B auf 1 und fragt sich dann wie viele Cs es geben kann. Also "Ein A und ein B haben x Cs".



Also "Ein Speiler spielt in einer Saison bei genau einem Verein". Heißt während einer Saison kann ein Spieler nicht den Verein Wechseln. Genau so gilt "In einer Saison sind in einem Verein beliebig viele Spieler" und "Ein Spieler Spielt in einem Verein für beliebig viele Saisonen". Die erste Eigenschaft kann man nur doch eine solche Beziehung Modellieren.

Assoziationsklassen: Wird durch eine gestrichelte Linie an eine Assoziation gebunden. Gibt noch mehr Informationen zu einer Assoziation. Also, wenn eine Person in einer Stadt wohnt, gibt, die Assoziationsklasse z.B. die Info an, von wann, bis wann. Dies ist dann nützlich wenn einer Person in beliebig vielen Städten wohnen kann. Dann kann man diese Information ja nicht einfach in die Person oder Stadt speichern - also neue Klasse.