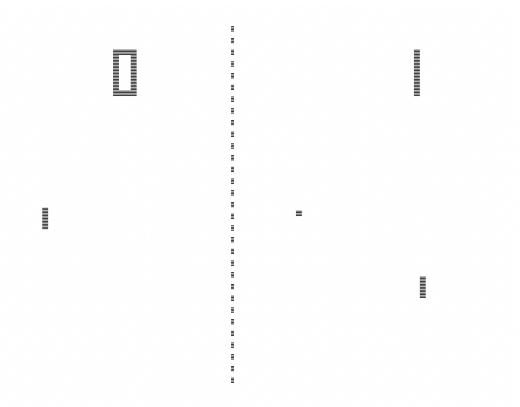
Schritt 1: Grundgerüst des Pong-Spiels (angelehnt an die Urversion von 1972)

Ziele

1. Anwenden der Klassen von TigerJython z.b. GameGrid,Actor
2. Erstellen und Benutzen eigener Klassen
3. Modellierung mit UML

Das Ur-Pong von 1972. Hier aus Tonerersparnis invertiert dargestellt:



Es lassen sich folgende **Objekte** identifizieren, die für unsere Modellierung wichtig sind:

- Ein Ball
- Zwei Schläger
- Eine Mittellinie – in dieser Version weggelassen
- Ein Punktezähler für jede Seite (Spieler) – erst im späteren Verlauf

Voraussetzungen

- Begriffe der OOP
- erste Vorübungen
- Erstellen von Klassen in Python und Instanziierung
- GameGrid - Hinweise zu Tastatur-Events und GameGrid- Zellen (PDF-Datei)
- TigerJython GameGrid Simulations Ablauf (PDF-Datei)
- OOPong-Schritt00 (PDF-Datei)

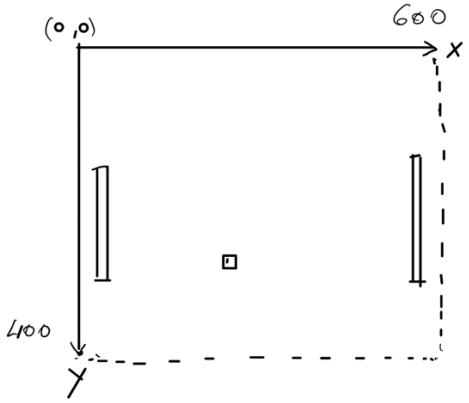
Was brauchen wir?

- Die Dokumentation zur Bibliothek GameGrid. Dort ist auch der import des Moduls beschrieben.
  Ein Spielfeld der Größe 600 x 400 (Breite x Höhe) - erzeugt durch makeGameGrid:

  makeGameGrid(600,400, 1,Color.black, False)
  show() # erst mit der methode show wird das Spielfeld angezeigt. vgl. weiter unten!

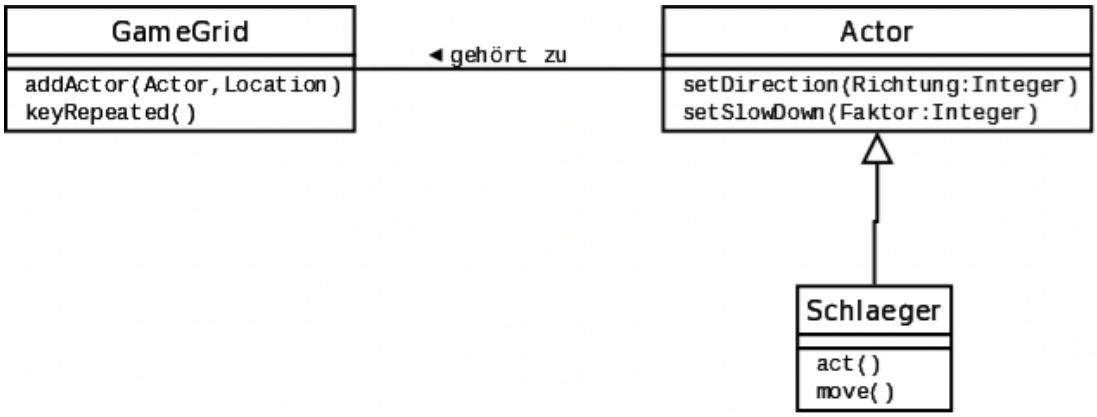
  erzeugt ein Spielfeld mit der angegebenen Größe, einer Grid-Größe von 1 und der Hintergrundfarbe Schwarz. Ein Gitter wird nicht dargestellt (False). Der Nullpunkt (0,0) liegt **links oben**.

- Zwei Schläger-Objekte: Repräsentiert durch eine Klasse *Schlaeger*.
  Bei Instanziierung (Aufruf des Konstruktors *__init__*) wird aus dem Unterverzeichnis *images* das zugehörige jpg-Bild des Schlägers zur Darstellung verwendet (vgl. Quelltext weiter unten).
  Hinweis 1: da TigerJython mit einem auf Java basierenden Python = Jython arbeitet, gibt es bei der Verwendung **eigener Bilder** die im Quelltext angeführten wichtigen Neuerungen zu beachten. Diese sind so nicht in der Dokumentation von Tobias Kohn zu finden! Die Verwendung der mitgelieferten Sprites folgt weiterhin der veröffentlichten Dokumentation
  Hinweis 2: das Verzeichnis *images* enthält alle notwendigen Bilder. Um dem beispielhaften Projektablauf zu folgen ist es notwendig, nur diese zu verwenden. Das Verzeichnis *images* muss sich immer im gleichen Ordner wie die Programmdateien (.py Dateien) befinden!



- Die Klasse Schlaeger erbt von Actor
- Eine UML-Darstellung. Siehe Aufgabe.
- Einen Lernpfad. Diesen hier. Er sollte nicht verlassen werden! Auch wenn die Geschichte von Hänsel und Gretel am Ende gut ausgeht. Der böse BUG lauert überall abseits des Weges.

Aufgabe:
Implementieren Sie das vorgegeben UML-Modell in einer Python-Datei. Diese enthält die Klasse(ndefinition) Schlaeger und einen main-Teil, der ein Schläger-Objekt erzeugt und verwaltet. Speichern Sie die Datei mit dem Namen *pongV01*. Erweiterungen oder Ergänzungen speichern Sie für diesen Schritt unter *pongV01a*, *pongV01b*, etc. **Hinweis**: wir werden ab Schritt03 die Klassen in eigene Dateien auslagern. Sie können dieses Vorgehen jetzt schon probieren, wenn Sie die Hinweise in Schritt03 gelesen und verstanden haben!



Hinweis: Die „gehört zu“-**Assoziation** wird durch den Aufruf der Methode *addActor* von GameGrid realisiert. Genaueres findet sich in der Dokumentation zu GameGrid bzw. Actor.

Hinweise zur Funktionalität

Die Schläger bewegen sich in Abhängigkeit der gesetzten *SimulationPeriod* immer auf und ab, jeweils links bzw. rechts am Spielfeldrand. *setSimulationPeriod(20)* setzte beispielsweise die Dauer eines Durchlaufs (Darstellen der Sprites, Ereignisbehandlung) auf 20 ms (vgl. GameGrid Simulations Ablauf).
Mit der von Actor geerbten Methode *setSlowDown(Faktor)* kann der Aufruf der *move()* Methode um einen ganzzahligen(!) Faktor verlangsamt werden. Beispiel:

schlaegerRechts.setSlowDown(3)

lässt 3 Simulationszyklen (im Beispiel á 20 ms) verstreichen, bevor die *move*() Methode des Actors aufgerufen wird. Hier erfolgt also alle 60 ms ein move() Aufruf.
Die Gesamtgeschwindigkeit lässt sich deshalb am besten mit diesen beiden Methoden fein einstellen:

setSimulationPeriod und *setSlowDown*

Die Schrittweite mit move() zu verändern ist zwar möglich, allerdings in diesem Spiel nicht sinnvoll. Probieren Sie verschiedene Werte für die o.a. Methoden aus, um ein Gefühl für das Zusammenspiel beider Methoden und die Spielgeschwindigkeit zu bekommen. Im Spiel lassen sich damit später sehr einfach variable Spielgeschwindigkeiten einrichten, z.B. indem der Ball nach einer bestimmten Spielzeit immer schneller wird.

Ereignis "Taste gedrückt"

Eine für das Ereignis „gedrückte Taste“ **vorhandene** Methode *keyRepeated* aus der Klasse GameGrid wird durch eine **eigene** Methode *keyCallback* überschrieben. Dies wird erreicht durch einen weiteren Parameter im Konstruktor von gamegrid:

```
makeGameGrid(600, 400, 1, Color.black, False, keyRepeated = keyCallback)
```

Dem Ereignis des wiederholten Tastendruckes (*keyRepeated*) wird die eigene Funktion *keyCallBack* zugewiesen. Diese kann auch anders benannt werden. Dann muss der Name *keyCallBack* natürlich bei dem Aufruf des Konstruktors von *makeGameGrid* ebenfalls geändert werden.

```
# Beginn Funktionsdefinition Tastenereignis
def keyCallback(keyCode):
    if keyCode == 87: # W nach oben
        schlaegerLinks.setDirection(270)
    if keyCode == 83: # S nach unten
        schlaegerLinks.setDirection(90)
    if keyCode == 73: # I nach oben
        schlaegerRechts.setDirection(270)
    if keyCode == 75: # K nach unten
        schlaegerRechts.setDirection(90)
# Ende der Funktion -------------
```

Die Simulation zum Laufen bringen

Am Ende unserer ersten Iteration[1] des Pong-Spiels stehen noch drei wichtige Funktionsaufrufe des GameGrid:

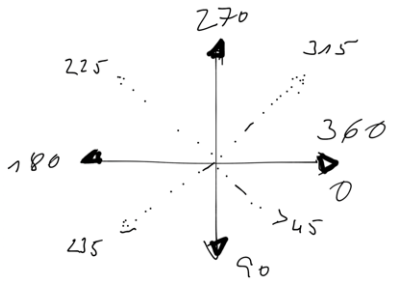
```
show() # zeigt das Fenster
setSimulationPeriod(10) # setzte die Simulationsperiode in ms
doRun() # startet die Simulationsschleife
```

Endergebnisse

1. GameGrid-Fenster mit Tastaturevents
2. Klasse Schläger mit Auf-und-Ab-Bewegung. Hinweis: zu Beginn unserer Programmierung arbeiten wir aus Vereinfachungsgründen noch mit der Integration der Klasse in das Hauptprogramm *pongV01.py*. Später werden wir jede Klasse in einer eigenen Datei auslagern.

```
1  from gamegrid import *
2
3  # ----------------- class Schlaeger -----------------
4  import os # WICHTIG, sonst funktioniert das Laden der eigenen Bilder nicht!
5
6  class Schlaeger(Actor):
7      def __init__(self):
8          dateiname = "images/schlaeger_small.jpg" # Dateiname MIT Unterordner
9          aktuellerPfad = os.path.abspath(".") # der Pfad zum Bild wird (intern) gebraucht und hier vom System ausgelesen
10         Actor.__init__(self, aktuellerPfad+"/"+dateiname) # der Pfad zum Bild für den Actor wird im Konstruktor benötigt
11
12     def act(self): # diese Methode wird periodisch aufgerufen
13
14         self.move()
15
16         # x- und y-Koordinate beziehen sich auf die Mitte des Sprites
17         if (self.getY() > 400): # falls Schlaeger unten aus Spielfeld,
18             self.setDirection(270)
19
20         if self.getY() < 0: # falls Schlaeger oben aus Spielfeld
21             self.setDirection(90)
22 # END of class Schlaeger -----------------
23
24
25 # ----------------- MAIN  (Hauptprogramm) -----------------
26
```

Richtungen (Direction) in TigerJython:



Hinweis: die x- und y-Koordinaten **eines Actor-Objektes** (*self.getY()* bzw. *self.getX()*) beziehen sich immer auf dessen **Mittelpunkt**. Bei der obigen Lösung verschwindet der Schläger somit immer um die Hälfte aus dem Spielfeld. Das ist beabsichtigt und vereinfacht das spätere Treffen des Balls.

[1] Iteration (von lateinisch iterare ‚wiederholen') beschreibt allgemein einen Prozess mehrfachen Wiederholens gleicher oder ähnlicher Handlungen zur Annäherung an eine Lösung oder ein bestimmtes Ziel. Hier sind die verschiedenen, aufeinander aufbauenden Programmversionen gemeint.