Schritt 5: Klasse CollisionsListenerBall

Erwartete Endergebnisse

- Die Klasse CollisionsListenerBall ist in einer eigenen Datei abgespeichert.
- main.py erzeugt ein CollisionsListener-Objekt für einen Ball
- Ball löst Kollision aus bei Berührung mit Schlägern

Hinweise

Die GameGrid-Bibliothek stellt uns eine Klasse ***GGActorCollisionListener*** zur Verfügung (vgl. Dokumentation dort). Folgendes ist am Anfang nicht leicht zu verstehen, stellt allerdings ein wichtiges Prinzip in der Objektorientierung dar. Wir **erben** von einer Klasse aus dem GameGrid und **überschreiben** bestimmte Methoden, damit diese das machen, was **wir** wollen (und nicht der Entwickler der GameGrid-Bibliothek). Oft sind diese Methoden auch gar nicht vom Entwickler ausprogrammiert, sondern nur als Gerüst vorhanden. Das Gerüst besteht aus der Spezifikation, indem der Methodenname und die notwendigen Parameter in der Dokumentation festgehalten sind. Mit diesem Wissen ist dann bei der eigenen Implementierung ein Überschreiben überhaupt erst möglich.

Man fragt sich sicherlich, warum die Kollision unbedingt mit der zugegeben nicht einfachen Methode über eine Kollisions-Klasse gelöst werden muss. *Ich brauche doch nur die Koordinaten der Objekte abzufragen*. Richtig! Das ist eben Unterricht. Da wir hier den Umgang mit Bibliotheken lernen wollen, ist es gerade notwendig, diesen Weg zu gehen. Hintergrund ist, dass bestimmte Dinge oft nicht so „einfach“ auf anderem Wege zu realisieren sind wie in diesem Beispiel die Kollision. Diese ist allerdings als Beispiel recht gut nachvollziehbar und griffig.

Wichtig ist auch, sich die Denkweise der Anmeldung der Kollisonsobjekte anzueignen. Dazu im Text unten mehr.

1. Zuerst muss diese Klasse in einer eigenen Klasse abgeleitet werden (**Vererbung**). Diese nennen **wir** sinnigerweise *CollisionListenerBall*, da es sich um die Kollisionsbehandlung für den Ball handelt. Wir könnten allerdings auch einen beliebig anderen für uns sinnvollen Namen wählen.

   ```
   class CollisionListenerBall(GGActorCollisionListener):
   ```

2. Dann wird die in der Elternklasse *GGActorCollisionListener* befindliche Methode (vgl. Dokumentation GameGrid) von uns in unserer Klasse CollisionListenerBall **überschrieben**:

   ```
   def collide(self, a, b): # wird ausgelöst, wenn die beiden registrierten
   Actors a und b aufeinandertreffen
   ```

3. In unserem Fall ersetzen wir a und b durch sinnvolle Namen:

   ```
   def collide(self, ball, schlaeger):
   ```

   wir werden gleich sehen, warum diese Reihenfolge sinnvoll (und wichtig) ist.

4. In der main.py müssen nun folgende Schritte in der **richtigen** Reihenfolge durchgeführt werden:

   ```
   # erzeuge den CollisionListener um eine Kollision abzufangen
   ballCollisionListener = CollisionListenerBall()
   ```

   Dies erzeugt ein CollisionsListener-Objekt . Wir benennen es gleich so, dass der Bezug zum Ball deutlich wird.

5. Es müssen im weiteren Schritt alle Kollisionspartner erstmal mit Hilfe der Actor-Methode *addActorCollisionListener* unseren *ballCollisionListener* kennen lernen („haben“), vgl. UML. Wir fügen somit unser *ballCollisionListener*-Objekt den Schlägern und dem Ball hinzu.

```
# Melde den ballCollisionsListener bei allen betroffen Actors an
schlaegerLinks.addActorCollisionListener(ballCollisionListener)
schlaegerRechts.addActorCollisionListener(ballCollisionListener)
ball.addActorCollisionListener(ballCollisionListener)
```

6. Als vorletzter Schritt muss jeder Kollisionspartner (Schläger-Objekt) dem *ball*-Objekt hinzugefügt werden, damit die Methode *collide* im CollisionListener überhaupt bei einer Kollision aufgerufen wird. Die Reihenfolge der Parameter für *collide (siehe oben)* hängt davon ab, ob ein Schläger-Objekt zum Ball (wie hier) oder ein Ball zum Schläger-Objekt (*schlaegerLinks.addCollisionActor(ball)*) hinzugefügt wird.

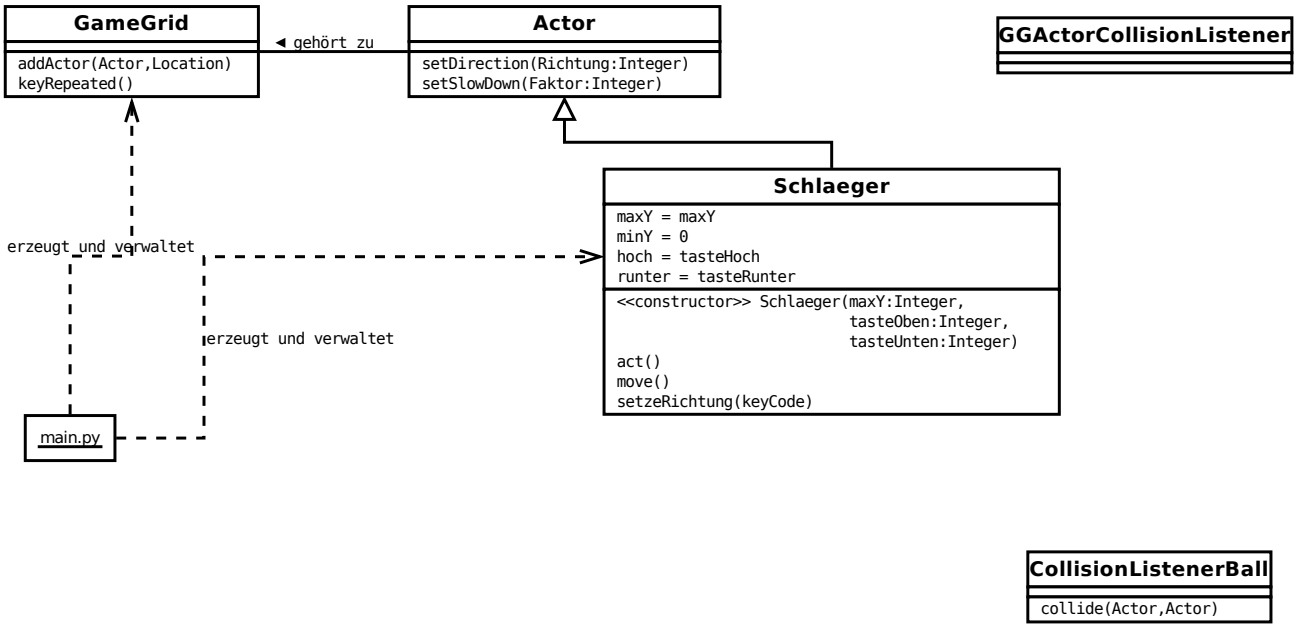
```
# def collide(self, ball, schlaeger) in CollisionListener (siehe oben!)
ball.addCollisionActor(schlaegerLinks)
ball.addCollisionActor(schlaegerRechts)

# alternativ hier eine andere Reihenfolge der Anmeldung
# def collide(self, schlaeger, ball) in CollisionListener
# dann gilt ABER
# schlaegerLinks.addCollisionActor(ball
# schlaegerRechts.addCollisionActor(ball)
```

7. Zuletzt muss die Methode collide in unserer CollisionListenerBall-Klasse noch implementiert werden:

```
# Callback beim Auftreten einer Kollision
def collide(self, ball, schlaeger): # wird ausgelöst, wenn die beiden re-
gistrierten Actors aufeinandertreffen
# hier steht das Programmstück für die Behandlung der Kollision zwischen
Ball und Schläger, das die Richtung des Balles ändert.
...# die Punkte stehen für die notwendige Implementierung
…
# WICHTIG ist das folgende return!
return 20   # gibt die Anzahl der Simulationszyklen zurück,
# während denen weitere Events unterdrückt werden.
# hier wählen wir sicherheitshalber 20
```

Aufgabe



Das abgebildete UML ist unvollständig. Ergänzen Sie die Klasse Ball sowie die fehlenden Beziehungen.

Ergänzung

Abschließend noch ein **Beispiel**, wie das Spiel einfach um einen zweiten Ball erweitert werden könnte. Probieren Sie es aus. Vielleicht können Sie bei einer späteren, eigenen, Version darauf als Vorlage zurückgreifen.

```
...
doRun()
# warte 5 Sekunden
delay(5000)
# erzeuge 2. Ball und setze diesen in die Mitte des Spielfeldes
ball2 = Ball(0, FENSTERBREITE, 0 , FENSTERHOEHE)
addActor(ball2, Location(FENSTERBREITE//2,FENSTERHOEHE//2)) # ACHTUNG! hier wird
reset von Actor aufgerufen, d.h. die Richtung wird auf 0 gesetzt!
ball2.setSlowDown(2) # Verlangsamt den Aufruf von move um den Faktor 2
ball2.setDirection(randint (0,360)) # setze den Ball auf eine zufällige
Anfangsrichtung
ball2.addActorCollisionListener(ballCollisionListener)
ball2.addCollisionActor(schlaegerLinks)
ball2.addCollisionActor(schlaegerRechts)
# nach 2 Sekunden wird der zweite Ball wieder entfernt
delay(2000)
removeActor(ball2)
```