Ergänzungsmaterial zum Lehrplannavigator

Eigene Klassen

In der vorliegenden Sequenz soll das Erstellen eigener Klassen in den Blickpunkt gerückt werden. Dazu gehört nicht nur die Implementation eigener Klassen in Java, sondern auch die Modellierung einer Problemstellung mit Hilfe eigener Klassen. Das Prinzip der Vererbung wird später behandelt und hier zunächst ausgespart.

Im Einzelnen sollen die folgenden Lernziele erreicht werden.

Lernziele

Die Schülerinnen und Schüler sollen …

- … eigene **Klassen** am einfachen Beispiel selbst implementieren können.
- … eigene **Methoden** implementieren können.
- … eigene **Parameter** in Konstruktoren und Methoden realisieren können.
- … eigene **Zustandsvariablen** umsetzen können.
- … die **Modellierung** einer einfachen Problemstellung mit Hilfe eigener Klassen im Sinne einer Komposition erstellen können.
- … das **Geheimnisprinzip** der objektorientierten Programmierung kennen lernen.

1 Sequenzskizze

Im Folgenden soll ein Einstieg in die Verwendung eigener Klassen mit Hilfe des Projektes *Kerzen* beschrieben werden. Das Projekt besteht darin, eine Simulation von drei brennenden Kerzen auf einem Holzbrett zu erstellen.

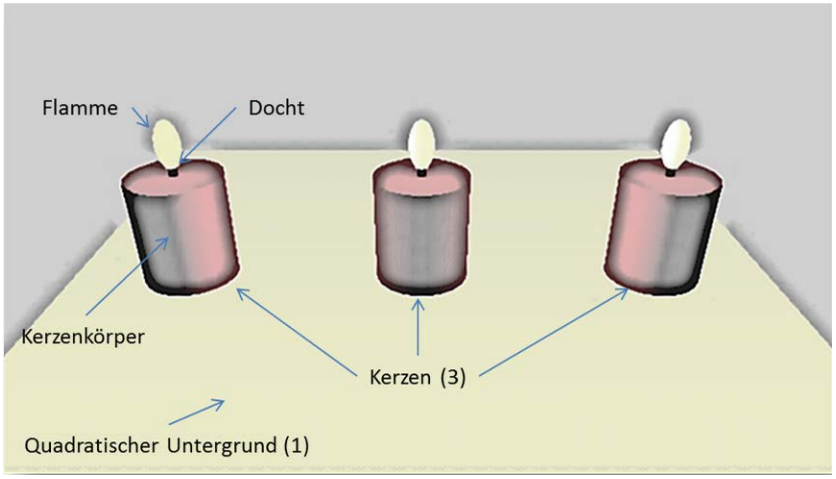

Abbildung 1: Planungsskizze zum Projekt *Kerzen*

Der Körper einer jeden Kerze besteht aus einem Objekt vom Typ `GLZylinder`, ebenso wie der jeweilige Docht einer Kerze. Die Flamme selbst ist ein Objekt vom Typ `GLLicht`, welches sich, sofern sichtbar geschaltet, als kleine Kugel zeigt. Diese kann mittels der Methode `skaliere()` zur endgültigen Flammenform verzerrt werden. Der Untergrund der Kerzen besteht aus einem mit Holztextur überzogenen Objekt vom Typ `GLQuader`.

Jede Kerze soll in der Simulation an- und ausgemacht werden können. Sofern sie gerade an ist, brennt sie ab, bis sie nur noch einen kurzen Stumpf darstellt und von selbst erlischt. Sie kann dann nicht mehr angezündet werden. Wie schnell eine Kerze abbrennt, ist abhängig von ihrer Dicke. Je dicker eine Kerze ist, umso langsamer brennt sie ab.

Auch bei diesem Projekt wird den Schülerinnen und Schülern wieder ein Prototyp vorgegeben, den es zunächst zu analysieren gilt. Im Gegensatz zu bekannten Programmen besteht dieser jedoch bereits aus drei rudimentären Klassen.

```
1  import GLOOP.*;
2  class Kerzenszene{
3      GLKamera kamera;
4
5      Kerzenszene(){
6          kamera = new GLKamera(800,600);
7          kamera.setzePosition(0,300,500);
8      }
9  }
```

```
1  import GLOOP.*;
2  class Untergrund{
3      GLQuader platte;
4
5      Untergrund(){
6          platte  = new GLQuader(0,0,-10, 600,20,600);
7          platte.setzeTextur("Holzboden.jpg");
8      }
9  }
```

```
1   import GLOOP.*;
2   class Kerze{
3       GLZylinder koerper, docht;
4       GLLicht flamme;
5
6       Kerze(){
7           //Kerzenkörper erstellen
8           koerper = new GLZylinder (0,50,0, 40,100);
9           koerper.drehe(90,0,0);
10          koerper.setzeFarbe(1,0,0);
11
12          //Docht erstellen
13          docht = new GLZylinder (0, 105, 0, 3,10);
14          docht.drehe(90,0,0);
15          docht.setzeFarbe(0,0,0);
16
17          //Flamme erstellen
18          flamme = new GLLicht(0,125,0);
19          flamme.setzeSichtbarkeit(true);
20          flamme.setzeFarbe(1,1,0.5);
21          flamme.skaliere(1,2,1);
22      }
23  }
```

Wird der Prototyp in der Entwicklungsumgebung *BlueJ* geladen, präsentiert er sich mit diesen drei Klassen, die in Form von drei Klassenkästen angezeigt werden. Da den Schülerinnen und Schülern Programme mit mehr als einer Klasse nicht vertraut sind, stellt sich zunächst die Frage, wie der Prototyp eigentlich zu starten ist. Erst die Instanziierung aller drei Klassen per Hand führt zum gewünschten Ergebnis und eine Kerze wird angezeigt.

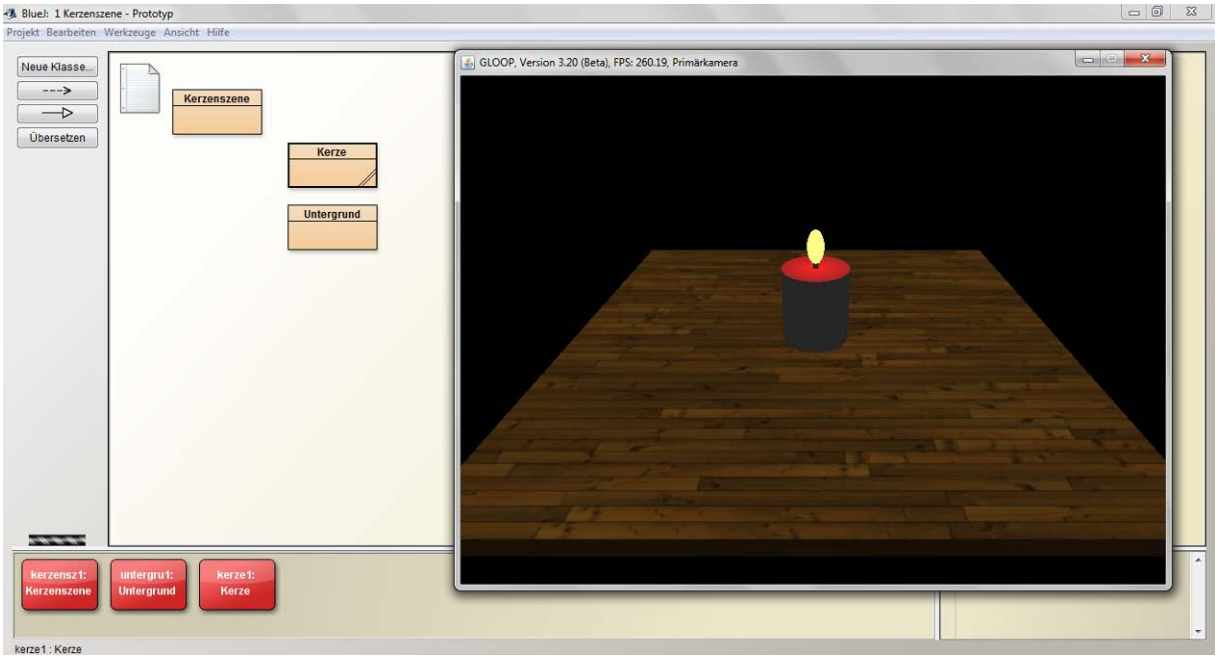

Abbildung 2: Laufender Prototyp zum Projekt *Kerzen*

Auf diese Weise demonstriert der Prototyp, dass alle in einem *BlueJ*-Projekt enthaltenen Klassen in einem Programmkontext zu sehen sind und gemeinsam eine lauffähige Simulation

ergeben können. Welche Objekte erstellt wurden, ist am linken unteren Ende des *BlueJ*-Fenstern zu sehen.

Der Prototyp liefert so zwar ein lauffähiges Ergebnis, ist von unseren Anforderungen aber noch weit entfernt. Es wird nur eine Kerze gezeigt, diese lässt sich nicht ausmachen und brennt auch nicht ab.

Mehr als eine Kerze zu erstellen, sollte jedoch kein Problem sein. Intuitiv könnte man auf den Gedanken kommen, dass einfach nur ein weiteres Objekt vom Typ `Kerze` erzeugt werden muss. Führt der Kursteilnehmer diesen Plan aus, kommt es zu einem interessanten Ergebnis. Es erscheint keine weitere Kerze im Bild, das Bild wird aber heller. Bei genauerer Überlegung wird schnell klar, woran das liegt. Es erscheint nämlich doch eine weitere Kerze im Bild, welche die Helligkeit der Szene durch ihre Flamme erhöht. Sie befindet sich aber an exakt der gleichen Stelle wie die erste Kerze und ist daher optisch nicht als separates Objekt wahrzunehmen. Mit diesem Phänomen als Motivation lassen sich nun Zustandsvariablen, Parameter und Methoden einführen. Eine ausführlichere Modellierung des Projektes *Kerzen* könnte dann wie folgt aussehen.

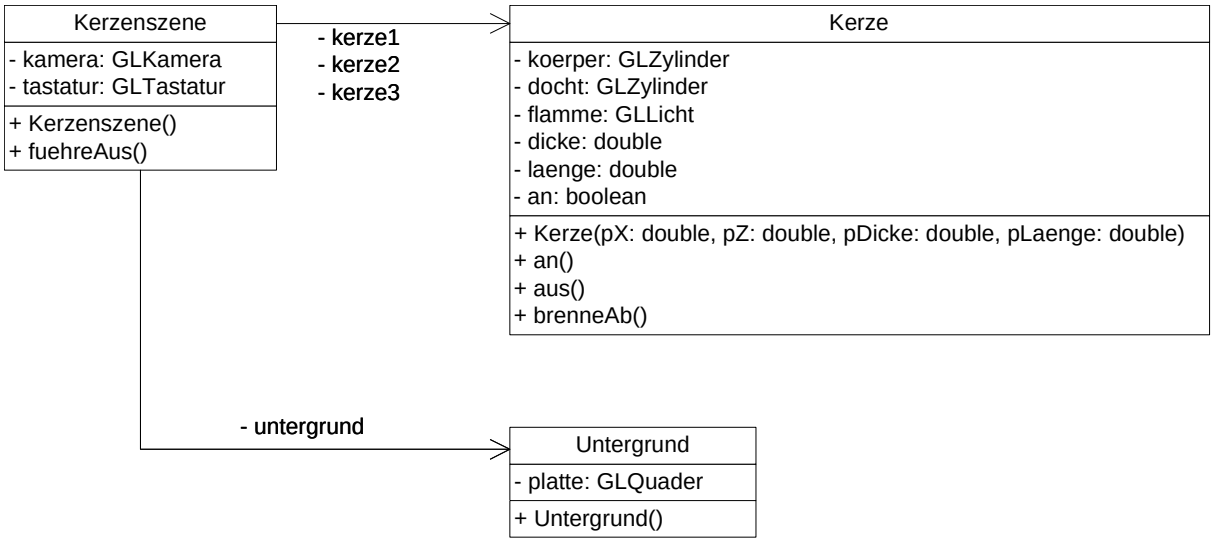

Abbildung 3: Modellierung des Projekts *Kerzen*

Im Konstruktor der Klasse `Kerzenszene` werden Untergrund und Kerzen erstellt. Die Methode `fuehreAus()` beinhaltet eine Animationsschleife, in der mit Hilfe einer Tastaturabfrage die Methoden `an()` bzw. `aus()` der einzelnen Kerzen aufgerufen werden können. Des Weiteren wird für jede Kerze die Methode `brenneAb()` aufgerufen.

Die Kerzen selbst bestehen aus einer Komposition von zwei Zylindern und einer Lichtquelle. Die beiden Zylinder stellen den Kerzenkörper und den Docht dar, während die Lichtquelle für die Flamme der Kerze steht.

Die Methoden `an()` bzw. `aus()` lassen die Lichtquelle erscheinen bzw. verschwinden, wohingegen die Methode `brenneAb()` abhängig von der Dicke der Kerze und der Information, ob die Kerze gerade an ist, ihre Länge mittels Skalierung und Verschiebung reduziert. Sie lässt, wie der Name schon sagt, die Kerze abbrennen, bis sie eine Minimallänge unterschreitet und dann ausgemacht wird.

Um das zu realisieren, muss die Kerze ihren eigenen Zustand in einer Reihe von Variablen nachhalten. Dicke, Länge und die Information, ob sie gerade an ist, werden in entsprechenden Attributen gespeichert und von den Methoden verwendet. Die Position der Kerze wird zwar im Konstruktor übergeben, so dass nicht alle Kerzen an derselben Stelle zu sehen sind, sie wird aber nicht in neuen Zustandsvariablen gespeichert. Die Position der Kerze ist nur dann von Bedeutung, wenn die *GLObjekte* erstellt werden, aus denen die Kerze besteht. Zu einem späteren Zeitpunkt muss diese Position nicht mehr abgefragt werden, so dass keine Notwendigkeit besteht, sie in Zustandsvariablen nachzuhalten.

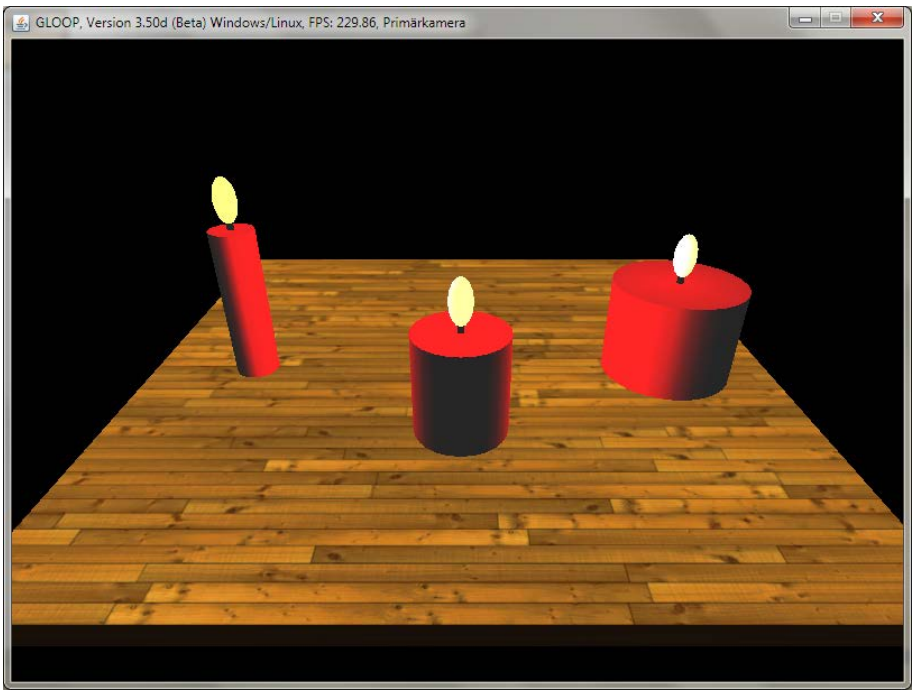

Abbildung 4: Laufende Kerzenszene mit drei Kerzen

Darüber hinaus lässt sich an diesem Beispiel das Geheimnisprinzip der objektorientierten Programmierung verdeutlichen. Könnte man aus der Klasse Kerzenszene auf das Attribut `an`

zugreifen, so könnte man die Kerze zum Abbrennen bringen, obwohl gar keine Flamme zu sehen ist und auf diese Art die Funktionsweise eines Kerzenobjekts kompromittieren. Die Attribute der Klasse Kerze sollten also geschützt werden, indem ein Zugriff von außerhalb der Klasse verweigert wird. Ist dieser Mechanismus einmal eingeführt, sollte er im weiteren Verlauf des Moduls bei allen Attributen und Methoden zum Einsatz gebracht werden, indem diese als *privat* oder *öffentlich* deklariert werden.

Der folgende Quellcode stellt die Methode `brenneAb()` der Klasse Kerze dar:

```
1       public void brenneAb(){
2           if (an) {
3               //Abbrenngeschwindigkeit in Abhängigkeit von der Dicke
4               double g =  (50/dicke) /1000;
5
6               //Kerzenkoerper durch Skalierung verkleinern
7               koerper.skaliere(1, 1, 1.0 - (g/laenge));
8
9               //Bestandteile verschieben
10              koerper.verschiebe(0,-g/2,0);
11              docht.verschiebe(0,-g,0);
12              flamme.verschiebe(0,-g,0);
13
14              //Attribut anpassen
15              laenge = laenge-g;
16          }
17
18          //Kerze geht ggf. aus.
19          if (laenge < 20) {
20              this.aus();
21          }
22      }
```

Die Methode `brenneAb()` zeigt sehr zutreffend den Charakter eines Dienstes im Sinne der objektorientierten Programmierung. Zwar stellt die Methode den Auftrag dar abzubrennen, die Kerze zeigt aber ein, wenn auch rudimentäres, selbstgesteuertes Verhalten und brennt nur dann ab, wenn die Kerze auch an ist. Objekte, die innerhalb klar definierter Grenzen autonom Aufträge interpretieren, sind typisch für die objektorientierte Programmierung.

Eine geringfügige mathematische Schwierigkeit stellt die Längenreduktion der Kerze dar. Die lokale Variable `g` errechnet die Längenreduktion in Abhängigkeit von der Kerzendicke.

Das Attribut `laenge` wird mit `laenge = laenge – g` entsprechend angepasst. Die optische Verkürzung der Kerze wird über den Dienst `skaliere()` der Klasse `GLObjekt` durchgeführt. Zur Verkürzung der Länge des Zylinders `koerper` um den Wert `g` muss also ein entsprechender Skalierungsfaktor errechnet werden. Dieser Skalierungsfaktor ergibt sich

durch den Term `1.0 - (g / laenge)`. Diese Umrechnung kann vermieden werden, indem von Anfang an mit einem Skalierungsfaktor gearbeitet wird. In dem Fall wird die Kerze aber nur schwer so zu realisieren sein, dass sie gleichmäßig abbrennt. [1]

2 Vertiefung

Um die am Kerzenbeispiel eingeführten Inhalte und insbesondere die Modellierung mit Hilfe eigener Klassen einzuüben, bietet sich das Projekt *Analoguhr* an. Bei diesem Vorhaben soll eine Szene erstellt werden, in der drei Uhren mit unterschiedlichen Zeitzonen angezeigt werden.

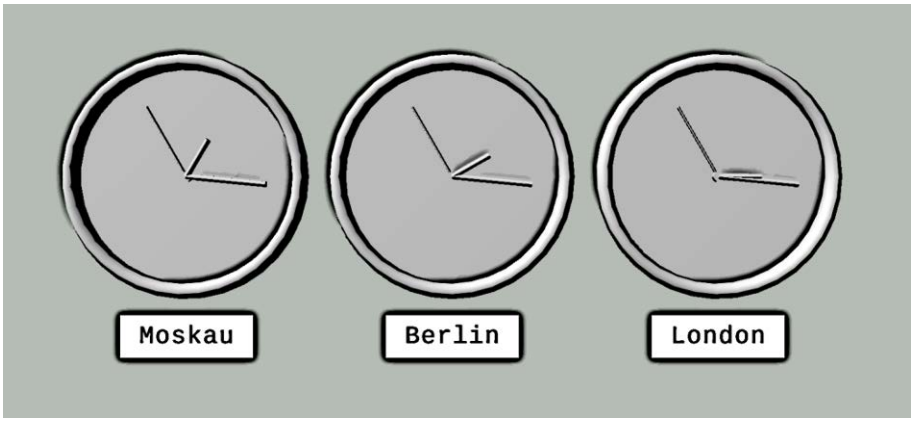

Abbildung 3: Planungsskizze zum Projekt *Analoguhren*

Um dieses Projekt zu realisieren, sind unzählige verschiedene Modellierungen möglich, so dass Schülerinnen und Schüler einen Eindruck vom kreativen Aspekt der informatischen Modellierung bekommen können. Verschiedene Modellierungen können erstellt, verglichen und bewertet werden, wobei nicht nur implementationstechnische, sondern auch logische Bewertungskriterien Berücksichtigung finden sollten.[2]

Eine mögliche Modellierung dieses Projektes soll im Folgenden dargestellt werden:

[1] Vergleiche hierzu auch die GLOOP-Dokumentation zum Dienst `skaliere`.
[2] Gemeint ist hier, dass eine Modellierung ohne z.B. die Klasse Zeiger womöglich effizient umzusetzen ist, aber aus einer Perspektive der Modellierung heraus dennoch zu kritisieren ist.

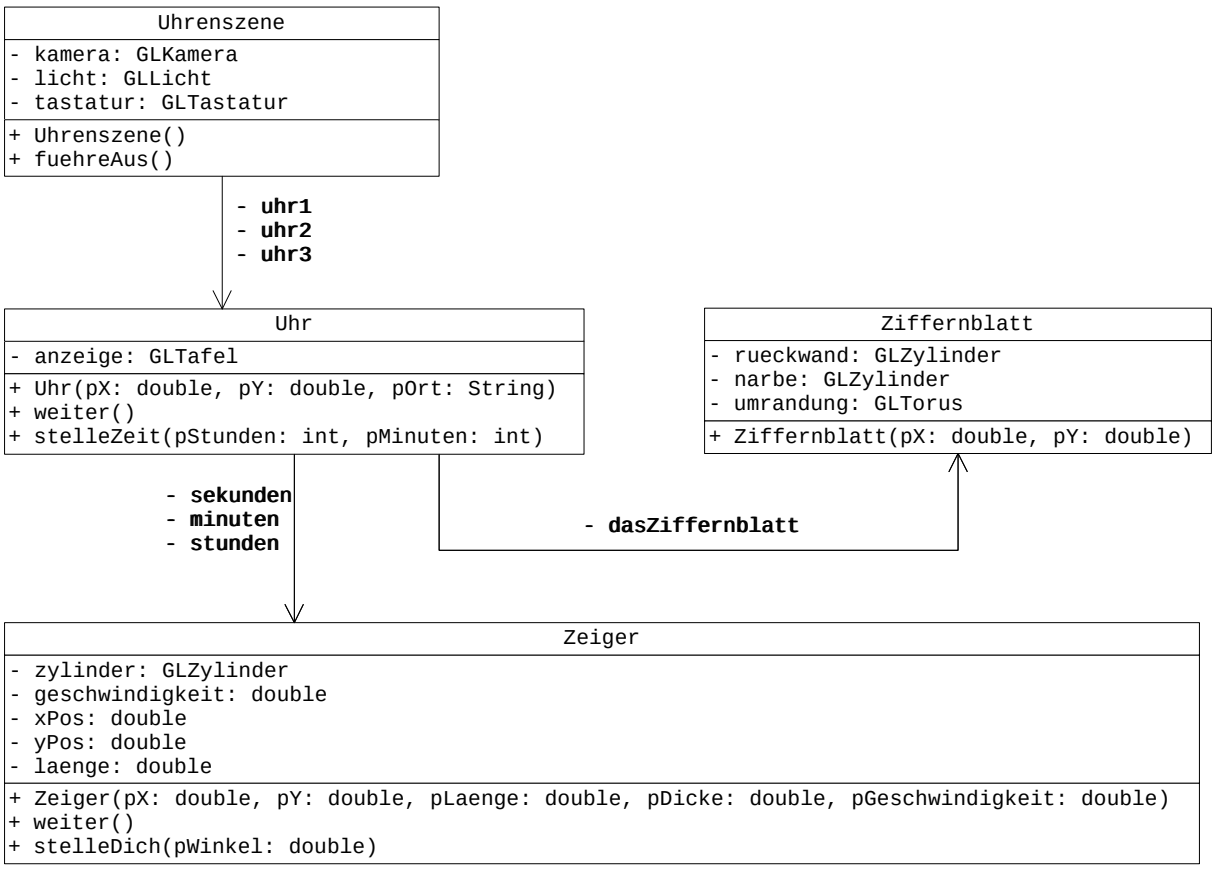

Abbildung 4: Implementationsdiagramm zum Projekt *Analoguhren*

Die Klasse Uhrenszene erstellt drei Uhren, stellt ihre Zeit und verfügt über eine Animationsschleife, die einmal pro Sekunde die Methode `weiter()` aller drei Uhren aufruft.

Beim Erstellen einer Uhr muss deren Position in der XY-Ebene und der Name des Ortes angegeben werden, der als Schriftzug unter der Uhr erscheinen soll.

Die Uhr erstellt daraufhin ihr Ziffernblatt und ihre Anzeigetafel mit dem entsprechenden Schriftzug. Beide Elemente sind statisch und werden sich im weiteren Verlauf der Simulation daher nicht verändern. Des Weiteren werden drei Objekte vom Typ `Zeiger` erstellt: der Sekunden-, Minuten- und Stundenzeiger. Sie erhalten unterschiedliche Längen und Dicken, so dass sie voneinander zu unterscheiden sind. Darüber hinaus wird ihnen eine Geschwindigkeit mitgegeben. Entsprechende Werte werden in Zustandsvariablen gespeichert.

Die Methode `stelleZeit()` der Klasse `Uhr` und die Methode `stelleDich()` der Klasse `Zeiger` sind zum direkten Setzen der Uhr beim Programmstart gedacht.

Bekommt die Uhr nun den Auftrag `weiter()`, so ruft sie die gleichlautenden Methoden ihrer drei Zeiger auf. Diese werden sich dann entsprechend ihrer in Attributen gespeicherten Geschwindigkeiten drehen. Der Sekundenzeiger wird 60 Aufrufe dieser Methode benötigen,

um einmal über das Ziffernblatt zu wandern, der Minutenzeiger `3600 (60*60)` und der Stundenzeiger `43200 (12*60*60).`

Zu bedenken ist bei dieser Modellierung, dass sowohl die Uhr als auch die Zeiger über keine Attribute zum Speichern der aktuellen Zeit verfügen. Die Uhrzeit ergibt sich lediglich aus der Stellung der Zeiger auf dem Ziffernblatt. Man kann argumentieren, dass dies aus der Perspektive der Modellierung unglücklich ist, da jede Uhr im Zustand einer bestimmten Zeitanzeige ist und somit auch entsprechende Zustandsvariablen haben sollte. Andererseits ergibt sich auch bei einer realen Analoguhr die Uhrzeit allein aus der Stellung der Zeiger, ohne dass sie an einer anderen Stelle gespeichert wird.

Ist die Modellierung erstellt, erweist sich die Implementierung in den meisten Fällen als recht einfach. Der folgende Quellcode stellt die Klasse Zeiger dar. Sie ist nach obiger Modellierung die komplexeste Klasse des Projektes:

```
1   import GLOOP.*;
2   public class Zeiger{
3       private GLZylinder zylinder;
4       private double geschwindigkeit;
5       private double xPos, yPos, laenge;
6
7       public Zeiger(double pX, double pY, double pLaenge,
8                                       double pDicke, double pGeschwindigkeit){
9           zylinder = new GLZylinder(pX,pY+pLaenge/2,0,pDicke,pLaenge);
10          zylinder.drehe(90,0,0);
11          geschwindigkeit = pGeschwindigkeit;
12          xPos = pX;
13          yPos = pY;
14          laenge = pLaenge;
15      }
16
17      public void weiter(){
18          zylinder.drehe(0,0,-geschwindigkeit, xPos,yPos,0);
19      }
20
21      public void stelleDich(double pWinkel){
22          zylinder.setzeDrehung(90,0,0);
23          zylinder.setzePosition (xPos,yPos+laenge/2,0);
24          zylinder.drehe(0,0,-pWinkel, xPos, yPos,0);
25      }
26  }
```

3 Materialien

1. P06_Kerzenszene_EigeneKlassen
2. P07_Analoguhr_EigeneKlassen