Erweiterungen des Anfangsprojektes

Die Schülerinnen und Schüler bekommen das Anfangsprojekt mit allen Mängeln. Allerdings stellt sich die Frage, ob das Anfangsprojekt einen einfachen **Konstruktor** enthalten sollte, hier beispielsweise für die Tischklasse:

```
class Tisch:
    def __init__(self, sichtbar=False):
        """einfacher Konstruktor"""
        self.x=70
        self.y=10
        self.b=120
        self.t=60
        self.w=0
        self.f='red'
        self.s=sichtbar
        if sichtbar: self.Zeige()
```

Der Aufruf des Konstruktors, also das Erzeugen eines Objektes, erfolgt durch **`Tisch()`** oder **`Tisch(True)`**. Verwendet man diese Variante, hat man den Vorteil, dass die Schülerinnen und Schüler bei der folgenden Aufgabe vermutlich selbst auf die Forderung nach einer anderen Variante für den Konstruktor kommen.

Aufgabe 1:

- Erstelle eine Tischgruppe aus einem Tisch mit sechs Stühlen.
- Beobachte dabei, was bei der Arbeit an dieser Aufgabe auf unbefriedigende Funktionalität hindeutet.
- Formuliere Anforderungen an die Funktionalität.
- Versuche die Klassen um diese Funktionalität zu erweitern.

Funktionalität verbessern?

Das Verbessern der Funktionalität einer Anwendung ist eine keineswegs triviale Aufgabe. Dabei gilt: Mehr Funktionalität hat nicht notwendig auch ein erfolgreicheres Arbeiten zur Folge. Grundsätzlich sollte man bei der Entwicklung von Software notwendige Funktionalität anbieten, aber nicht alle möglichen Aufgaben berücksichtigen, die der künftige Nutzer haben könnte

Mängel des Projektes

Dennoch lässt sich für die vorliegende Anwendung feststellen: Die Klassen Stuhl und Tisch haben für den Benutzer unzureichende Funktionalität.
Erzeugt man nämlich mehrere Exemplare dieser Klasse und stellt sie dar, erhält man das unbefriedigende Ergebnis, dass alle – in ihren Eigenschaften gleichen – Exemplare sich überdecken und wie eines erscheinen, also optisch nicht unterscheidbar sind, obwohl sie selbstverständlich nicht dieselben Objekte sind.
Allgemein gilt, dass verschiedene Objekte durchaus in all ihren Attributwerten übereinstimmen dürfen, ohne dass sie damit dasselbe Objekt sind. Gerade bei Möbelstücken kann das an sich sinnvoll sein, wenn es beispielsweise nur einen Typ gibt. Allerdings stehen sie aber sicher nicht alle am selben Platz und sollten daher in diesen Attributwerten verschieden sein. Daher wäre es vorteilhaft, bereits beim Erstellen von neuen Stühlen und Tischen die Position, Größe, usw. beeinflussen zu können.

Eine Alternative für den Konstruktor

Ein Konstruktor, der diese Möglichkeit bereitstellt, dem dazu die Parameter übergeben werden müssen, sieht dann so aus:

```
    def __init__(self, xPos, yPos, breite, tiefe, winkel, farbe, sichtbar):
        """Konstruktor mit zu übergebenden Parametern"""
        self.x=xPos
        self.y=yPos
        self.b=breite
        self.t=tiefe
        self.w=winkel
        self.f=farbe
        self.s=sichtbar
        if sichtbar: self.Zeige()
```

Diese Variante bietet die Möglichkeit, bei guter Planung der Darstellung die Tischgruppe gleich durch den passenden Aufruf der Konstruktoren wie gewollt zu erstellen.

Ein vielseitiger Konstruktor statt vieler Konstruktoren

Flexibler wäre eine Lösung, die in der Klasse einen Konstruktor anbietet, der zwar dieselben Standardwerte für alle Attribute vorsieht, aber die Übergabe gewünschter Attributwerte zulässt. Bei Java wird dies durch das Einführen mehrerer Konstruktoren gelöst, die sich in ihrer ***Signatur*** unterscheiden, also der Anzahl der Parameter oder der Reihenfolge der Parametertypen. Man bezeichnet das Vorgehen als ***overload***, das von Java auch für andere Methoden angeboten wird.
Die Entwickler von Python sind einen anderen Weg gegangen. Python bietet standard-mäßig die Möglichkeit, Methoden mit vordefinierten Parametern zu verwenden. Dazu wird im Kopf der Methode den Parameternamen ein Wert zugewiesen. Bei einem Aufruf, der diesen Parameter nicht neu definiert, wird also der Standardwert verwendet.
Die Zuweisung von Werten zu den Parametern beim Aufruf des Konstruktors kann auf zwei Arten geschehen. Werden die Parameter ohne ihre Bezeichnung übergeben, werden die Werte den Parameter in der Reihenfolge ihres Auftretens zugewiesen. Folgen danach im Aufruf Zuweisungen mit Namen, also z.B. **`farbe='rot'`**, dann werden sie unabhängig von der Reihenfolge den richtigen Parametern zugewiesen.

```
    def __init__(self,
                 xPos=70,
                 yPos=10,
                 breite=120,
                 tiefe=60,
                 winkel=0,
                 farbe='red',
                 sichtbar=False):
        """Konstruktor mit vordefinierten Parametern"""
        self.x=xPos
        self.y=yPos
        self.b=breite
        self.t=tiefe
        self.w=winkel
        self.f=farbe
        self.s=sichtbar
        if sichtbar: self.Zeige()
```

Attributnamen und Methodennamen

Die Klasse Stuhl verwendet für das Speichern der Eigenschaften (***Attribute***) `xPos, yPos, breite, tiefe, winkel, farbe, sichtbar` intern die Felder (Variablen) x, y, f, b, t, w, f, und s. Bei den Parameternamen finden wir im Projekt eine vielfach verwendete Schreibweise: Attributnamen beginnen mit einem kleinen Buchstaben. Setzt sich der Name aus mehreren Worten zusammen, werden sie zusammenhängend geschrieben, damit das System nicht meint, es seien mehrere Variable gemeint und der Anfangsbuchstabe der nachfolgenden Teilworte wird jeweils mitten im Wort groß geschrieben [Kamel-Notation][1].
Bei Namen von Methoden ist es in Python selbst üblich Methodennamen klein zu schreiben. Da bei wxPython die Namen am Anfang groß geschrieben werden, habe ich im Projekt auch diese Schreibweise gewählt.

Erweiterung der Funktionalität durch weitere Methoden

Für weitere Methoden lassen sich viele Beispiele finden. Man könnte beispielsweise für den Raum ein Raster vorgeben, an dem sich alle Moebelstücke orientieren müssen. Ein Verschieben wäre dann nur um Vielfache dieses Rasters möglich, so dass man z.B. einmal oder mehrmals **`BewegeHorizontal()`** benutzt, ohne einen Wert zu übergeben und dann jeweils um eine vorgegebene Weite weitergesetzt wird.
Wichtig ist an dieser Stelle, dass die Schülerinnen und Schüler lernen, mit IDLE, Shell, PyShell und den Grundelementen der Sprache Python und wxPython umzugehen. Neue Sprachelemente werden dann besprochen, wenn man auf sie trifft.

Aufgabe 2

- Schritt 1: → Arbeitsblatt *Aufgabe zum Erstellen eines Sessels.*

- Schritt 2: Erstelle weitere Möbelklassen mit einem Erscheinungsbild, das den Normzeichen entspricht: Schrank, Bett, Regal, Sessel, Sofa, usw.

1 Die Entwickler von Python schreiben Namen allerdings häufig mit Unterstreichungsstrich zwischen den Namensteilen.

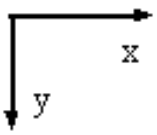
Das Zeichnen weiterer Möbelsymbole[1]

In unserem Raumplaner fehlen noch viele Klassen. Wenn wir sicher auch nicht Vollständigkeit anstreben werden, so geht es uns aber darum, die grundsätzlichen Konzepte des Zeichnens in Python zu verstehen. In unseren beiden vorliegenden Klassen sind zwei etwas voneinander abweichende Konzepte realisiert, die wir uns ansehen müssen. Beginnen wir mit der Klasse Stuhl. Im Programmtext finden wir eine Methode **`gibFigur()`**, die ganz offensichtlich die Zeichnung definiert:

```
def GibFigur(self, gc=None):
    path = Zeichenflaeche.GibZeichenflaeche().GibGC().CreatePath()
    path.MoveToPoint(0, 0)
    path.AddLineToPoint(self.b, 0)
    path.AddLineToPoint(self.b*1.1, self.t)
    path.AddLineToPoint(-self.b*0.1, self.t)
    path.AddLineToPoint(0, 0)
    path.AddLineToPoint(0, -self.t*0.1)
    path.AddLineToPoint(self.b, -self.t*0.1)
    path.AddLineToPoint(self.b, 0)
    gc = Zeichenflaeche.GibZeichenflaeche().GibGC()
    gc.PushState()
    gc.Translate(self.x+self.b/2, self.y+self.t/2)
    gc.Rotate(radians(self.w))
    gc.Translate(-self.b/2, -self.t/2)
    transformation = gc.GetTransform()
    gc.PopState()
    path.Transform(transformation)
    return path
```

Die Ähnlichkeit des ersten Methodenabschnittes zu Beschreibungen in LOGO – ähnlichen Systemen mit turtle - Grafik ist so deutlich, dass das Verständnis nicht schwer fallen sollte.

- **`MoveToPoint(xPos, yPos)`** ist eine Methode, die den Zeichenstift auf die durch (xPos, yPos) definierte Bildschirmposition *setzt*, ohne dass dabei gezeichnet wird.
- Das Grafiksystem verwaltet also offensichtlich die aktuelle Position des Zeichenstiftes.
- **`AddLineToPoint(xPos, yPos)`** ist eine Methode, die von dieser aktuellen Position des Zeichenstiftes des Zeichenstift eine Linie zur durch (xPos, yPos) definierten Bildschirmposition *zeichnet*.
- Positionen werden in Bildschirmkoordinaten angegeben. Das System beginnt in der linken oberen Ecke unserer Zeichenfläche im Punkt (0,0), die x – Richtung geht nach rechts und –ungewohnt– die y – Richtung nach unten.



Mit dem im zweiten Abschnitt des Programmtextes in der Methode **`GibFigur(...)`**, der mit der Zeile **`gc.PushState()`** beginnt, wollen wir uns zunächst noch nicht beschäftigen. Diesen betrachten wir zunächst genauso als eine „black box“, wie die Klasse Grafikfenster. Wir akzeptieren zunächst, dass sie das tun, was wir wollen und heben uns eine Klärung, wie das geschieht, für später auf.
Das Zeichnen eines Sessels, Sofas, Schrankelements oder Betts sollte nach dieser Kenntnis nicht schwer fallen.

1 Bild zu den Möbelsymbolen auf einem Extrablatt.

Einige Fragen bleiben aber:
- Wie erstelle ich die neue Klasse?
- Wie vermeide ich, dass der Programmtext völlig unübersichtlich wird?
- Wie wird beispielsweise ein runder Tisch gezeichnet?

Eine neue Klasse erstellen

Die neue Klasse erstellt man am einfachsten in zwei Schritten. Man kopiert den Programmtext der Klasse Stuhl und fügt ihn zum zweitenmal in das Projekt ein, um ihn anschließend passend zu bearbeiten.
Im vorgegebenen Anfangsprojekt sind die wichtigen Klassen auf verschiedene Dateien verteilt, obwohl das nicht notwendig ist. Hat man alle Klassen in einer Datei, sind keine Importe innerhalb des Projektes notwendig.

Programmtext übersichtlich halten

Je mehr neue Klassen wir aber in eine Datei einfügen, desto unübersichtlicher wird der Programmtext. Wir können endloses Blättern vermeiden, wenn wir die einzelnen Klassen in verschiedene Dateien ausgliedern, wie das im Anfangsprojekt gemacht worden ist.
Das Projekt ist aufgeteilt auf die Dateien stuhl.py, tisch.py, grafikfenster.py und raumplaner.py. Da die Klassen Stuhl und Tisch auf das Grafikfenster zugreifen, muss alles aus dem **Modul** grafikfenster durch **`from grafikfenster import *`** importiert[1] werden.

Namen ändern, auch in den Kommentaren

Bevor man etwas anderes macht, ändert man den Namen der Datei passend und in der Datei den Namen der Klassendefinition. Da unsere Projektdateien auch immer eine Anwendungsklasse enthalten, muss auch diese angepasst werden. Erst jetzt geht man daran, den Text der Methode **`gibFigur()`** zu korrigieren.

Nicht gradlinige Kurven

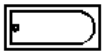
Die letzte Frage führt auf ein völlig neues Problem. Mit Hilfe von sehr vielen kleinen geraden Linienabschnitten könnte man natürlich das Problem lösen, eine gebogene Linie zu zeichnen. Unser Grafiktool wxPython bietet uns aber mit der Klasse **`wxGraphicsPath`** eine andere Möglichkeit, die wir hier nutzen. Dazu sehen wir in der Dokumentation nach und finden bei der Klasse wxGraphicsPath die passende Methode:

wxGraphicsPath::AddEllipse
void AddEllipse(wxDouble x, wxDouble y, wxDouble w, wxDouble h)
Appends an ellipse fitting into the passed in rectangle.

1 Import-Varianten im Anhang

Anhang: Ein Beispiel für einen weiteren Möbeltyp

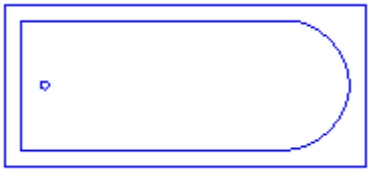
Einige der Möbelzeichnungen lassen sich nur aufwändig aus mehreren Teilfiguren zusammensetzen. Für solche Figuren ist die Klasse wxGraphicsPath sehr hilfreich, da sie nicht nur die elementare Zeichenmethoden wie **`moveToPoint(...)`** und **`AddLineToPoint(...)`** bereitstellt, sondern weitere.
Als Beispiel verwenden wir die Badewanne, die man aus mehreren Teilfiguren zusammensetzen kann, nämlich



- einem vollständigen Rechteck,
- einem unvollständigen Rechteck, das wir aus drei Linien mit einem Halbkreis (AddArc) daran zusammensetzen werden
- und einem kleinen Kreis (AddCircle) für den Ablauf.

Einige Tipps

Für die zu verwendenden Maße nehmen wir relative Werte, also Angaben, die abhängig von den Werten breite und tiefe sind. Vorgegebene Maße aus der Normzeichnung sind 180 x 80, das definiert uns die Werte für das volle Rechteck. Für das innere, unvollständige Rechteck lassen wir auf jeder Seite gleich viel, z.B. ca. 10% der Tiefe weg, so dass dessen Position in der Figur nicht durch die linke obere Ecke (0;0), sondern durch (0.9*tiefe;0.9*tiefe) beschrieben wird, seine Tiefe durch 0.8*tiefe (oben und unten weniger!). Die Breite ist komplizierter zu bestimmen, da auf der rechten Seite auch der Radius des Halbkreises abgezogen werden muss.



Man muss bei einer solchen Figur sicher etwas probieren, bis man passende Werte findet.

Anhang: Import-Varianten

Import-Variante 1

`import grafikfenster`
importiert das Modul grafikfenster in das Projekt. Wollen wir auf irgendetwas aus diesem Modul zugreifen, müssen wir immer kennzeichnen, dass es aus dem Modul stammt. Das geschieht grundsätzlich durch

```
<Modulname>.<Objektname>
```

Auf die Klasse Zeichenflaeche greifen wir also jeweils zu durch

```
grafikfenster.Zeichenflaeche
```

Import-Variante 2

`from grafikfenster import Zeichenflaeche`
importiert aus dem Modul allein die Klasse Zeichenflaeche, während
`from grafikfenster import *`
alle Objekte aus dem Modul importiert.
Die zweite Variante ist etwas komfortabler, hat aber den Nachteil, dass es beim Import mehrerer Module zu Überschneidungen von Namen und damit zu Konflikten kommen kann.

pyc-Dateien

Wird das Projekt vom Modul raumplaner.py ausgeführt – was aus der Datei mit dem Application-Objekt heraus geschehen muss – dann werden zunächst die importierten Module übersetzt und das Ergebnis in eine gesonderte Datei mit der Namenserweiterung .pyc geschrieben. Bei nachfolgenden Programmstarts sieht Python nun erst zunächst nach, ob eine zugehörige .py-Datei neuer ist. Wenn das nicht der Fall ist, wird die übersetzte letzte Variante verwendet, was bei großen Projekten durchaus die Startzeit verringern kann.