Das Raumplaner-Projekt entwickeln

Entwicklungsumgebung

Anders als bei der Arbeit mit Java, wobei uns mit dem Tool BlueJ eine Entwicklungsumgebung zur Verfügung steht, die gezielt für die Arbeit in der Ausbildung entwickelt wurde, arbeiten wir bei Python mit Standard-Entwicklungsumgebungen. Davon gibt es eine größere Zahl, wir werden im Kurs aber nur IDLE benutzen. IDLE ist die IDE, die unter Windows automatisch mit Python installiert wird, unter Linux muss man sie ggf. nachträglich installieren.

Grafik-Toolkit wxPython

Für die grafische Darstellung und vor allem auch für die Entwicklung von grafischen Benutzeroberflächen benötigen wir zusätzlich ein Grafik-Toolkit. Dafür gibt es mehrere Varianten, eine Erläuterung findet man im Python-Buch von Galileo-Computing[1]. Dort sind *Tkinter*, *PyGtk* und *PyQt* etwas ausführlicher beschrieben.
Bei allen Erläuterungen und Beispielen in diesem Kurs beziehe ich mich allein auf **wxPython**, da es zwei wichtige Kriterien erfüllt:

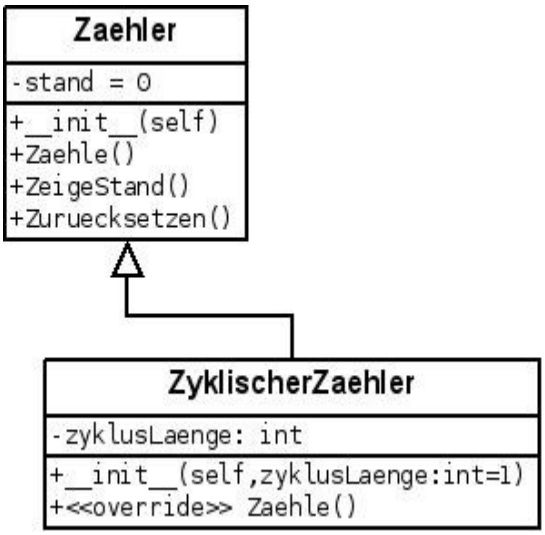
- wxPython ist hervorragend dokumentiert.
- Zu wxPython gibt es ein hervorragendes Demo-Tool, das zu den wesentlichen Aufgaben und Elementen von grafischen Oberflächen funktionierende und ausführlich dokumentierte Beispiele in einer eigenen Oberfläche bietet. Schülerinnen und Schüler können sich mit Hilfe dieser Beispiele [schon in der Mittelstufe] mit etwas Hilfe selbstständig einarbeiten.

Bei der Installation sollte man also darauf achten, dass man nicht nur wxPython installiert, sondern auch die **"docs, demos and tools"**.

UML zur Modellierung

Anders als bei BlueJ gibt es [kostenlos] keine integrierte UML-Umgebung [UML: **U**nified **M**odeling **L**anguage]. Um aus UML-Diagrammen Python-Code erstellen zu lassen [*"forward"*], habe ich bis auf ***DIA*** zusammen mit ***dia2code*** kein [kostenloses] Tool gefunden[2].

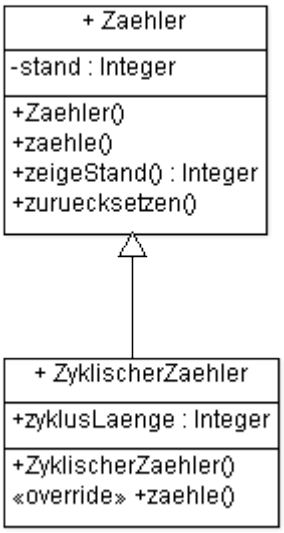
Unabhängig von der Frage, wie weit überhaupt Softwareentwicklung aus UML-Diagrammen heraus durchgeführt wird, setzt man die Diagramme in der Schule sinnvollerweise für die Modellierungsphase ein. Die Schülerinnen und Schüler müssen die Diagramme entwickeln, darstellen und interpretieren können.



DIA UML-Diagramm [Python-Sytax]

1 Leider ist Galileo-Computing an den Rheinwerk-Verlag übergegangen und man findet das Buch dort nicht mehr.

2 Mein Projekt zur Entwicklung von Klassendiagrammen kann das auf einfache Art.
Ein spannendes OO Projektthema wäre die Entwicklung einer Oberfläche, deren Möglichkeiten denen von BlueJ ähneln. Interaktives Arbeiten ist zwar mit dem ShellFrame möglich, es fehlt aber das integrierte, direkte Anzeigen von UML-Diagrammen zum Code.
Ein automatisch erstellter Beispielcode für das Raumplaner-Anfangsprojekt beispielhaft im Anhang.



Klassendiagramm ArgoUML [Java-Syntax]

ArgoUML

Die Entwickler von ArgoUML[1] haben Code-Generierung für Python geplant, sie bisher aber nicht umgesetzt. Insgesamt ist das Problem, dass die Entwicklung bei ArgoUML nicht weiter geht (V. 0.34).

Mein Projekt

Das von mir auf der Basis von Python mit wxPython entwickelte Projekt-Klassendiagramme kann eingesetzt werden. Es liefert neben den Diagrammen und der Möglichkeit in einfachen Pythoncode zu exportieren auch lesbare Modelldateien, was ich für den Einsatz im Informatikunterricht für interessant halte.

Python Package Index

Es gibt glücklicherweise zwar einige gute UML-Tools zu Python. Unglücklicherweise funktioniert deren Installation aber bisher leider nur bei Python2.7, also nicht bei den Python3-Versionen.
Ein neben DIA und ArgoUML für das Erstellen von Klassendiagrammen gut geeignetes Tool ist ***Gaphor***, das sich aus dem ***"Python Package Index PyPI"*** mit Hilfe von ***pip***[2] installieren lässt.

UML-Darstellung von vorhandenen Projekten

Es gibt auch Tools, die aus bestehendem Code die UML-Diagramme zeichnen [*"reverse"*]. Für die Darstellung der Klassendiagramme fertiger Projekte ist ***pyplantuml*** [3]gut geeignet (siehe Beispiel unten), allerdings muss man dabei mit der Kommandozeile arbeiten. ***Gaphor*** kann zwar Pythoncode importieren, die Ergebnisse sind aber etwas unbefriedigend, da in den Klassen die Attribute fehlen.

Am Start keine UML notwendig

Unser Anfangsprojekt ist so angelegt, dass am Beginn der Einsatz von UML nicht sinnvoll ist, da die Kursidee am learning by doing ausgerichtet ist.

Interaktives Arbeiten

Beim Starten von Programmen mit grafischen Oberflächen hat man das Problem, dass sie von einem Anwendungsobjekt (application) gestartet werden müssen, die Shell dann die Kontrolle voll an die Anwendung abgibt[4], so dass keine Eingabe mehr möglich ist und erst wieder aktiv wird, wenn die Anwendung beendet wurde.
Die besondere Möglichkeit von Python und wxPython mit dem ShellFrame nimmt die Idee des interaktiven[5] Arbeitens mit der laufenden Anwendung auf.

1 https://argouml-tigris-org.github.io/tigris/argouml/
2 pip sollte Teil der Python-Installation sein; anderenfalls sollte man es in jedem Fall nachinstallieren.
3 Installation ebenfalls mit Hilfe von pip aus PyPI.
4 Die REPL [read-eval-print-loop] bleibt im Zustand "eval", da die Ereignissteuerung der Anwendung aktiv bleibt.
5 Dazu gibt es die Klasse ShellFrame aus dem Paket wx.py.shell unter wxPython.

Objects first – BlueJ als Vorlage

Für einen Pythonkurs zur OO stellt sich die Frage, ob man das Objects first – Konzept, das Grundlage von BlueJ und dem BlueJ-Buch von Barnes und Kölling[1] ist, auch unter Python nutzbar machen kann. Die hier vorliegende Programmlösung zeigt neben den Erfolgen auch Schwierigkeiten auf, die dabei auftreten.

Eine einfache Lösung für den Raumplaner

Ein erster Ansatz zum Einstieg in das Raumplaner-Projekt nimmt die für Java mit BlueJ bereits entwickelte Version auf[2]. Das dort verwendete Anfangsprojekt arbeitet mit drei Klassen, einer Klasse Stuhl, einer Klasse Tisch und einer Klasse Leinwand, die leicht bearbeitet aus dem Projekt **`Figuren`** des BlueJ-Buchs übernommen worden ist.

Interessanterweise ergibt das Kopieren des unter Java entwickelten Codes für die Klassen Stuhl und Tisch in ein Python-Projekt zwar keinen funktionierenden Python-Code. Der Code kann aber sinnvoll strukturell übernommen werden. Daher kann es im Interesse des Vermittelns der Konzepte, die hinter der Syntax von Programmiersprachen stehen, sehr interessant sein, dies einmal wirklich am Beispiel einer der Klassen durchzuführen.

Wesentliche Änderungen ergeben sich allerdings bei der Klasse für die grafische Darstellung. Hier sind wegen etwas anderer Konzepte bei ***wxPython*** gegenüber Java merkliche Änderungen notwendig. Der Name der Datei zu dieser Klasse im Pythonprojekt ist **`grafikfenster.py`**, wie im unten angegebenen Anfangsprojekt aus der Importanweisung im Programmcode zu entnehmen ist.

Die enthaltene Klasse **`GrafikFenster`** (erbt von wx.Frame) verwendet die ebenfalls in der Datei enthaltene Klasse **`Zeichenflaeche`** (erbt von wx.Panel)). Sie ist nicht als interne Klasse realisiert wie beim Java-Projekt.
Die in den Anfangsprojekten verwendete Klasse **`Zeichenflaeche`** stellt nur Umrisse von Figuren dar. Es ist aber auch möglich Flächen farblich zu füllen .
Die Klasse **`GrafikFenster`** stellt zusätzlich Methoden zum Erzeugen des oben bereits erwähnten ShellFrames und des FillingFrames bereit, sowie eine Methode zum Export von Bildern des Grafikfensters.

1 Barnes, Kölling: Java lernen mit BlueJ ISBN 3-8273-7152-X
2 Siehe dazu das Material auf http://www.informatik-hamburg.de/

Der Programmtext von Stuhl und Tisch

```
### ---------------- Stuhl ----------------------------------
from grafikfenster import *
from math import radians

class Stuhl():
    """Klasse Stuhl
    ermoeglicht das Zeichnen und Bearbeiten eines
    Stuhl-Symbols fuer den Raumplaner"""

    def __init__(self,
                 xPos=20,
                 yPos=20,
                 breite=40,
                 tiefe=40,
                 winkel=0,
                 farbe="blue",
                 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()

    def GibFigur(self):
        """definiert und transformiert die zu zeichnende Figur"""
        gc = Zeichenflaeche.GibZeichenflaeche().GibGC()
        path = gc.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.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

    def GibFarbe(self):
        """Get-Methode fuer die Farbe"""
        return self.f
```

Import von - selbst geschriebenen und - aus Python benötigten Paketen

Kommentar

Konstruktor

Das geht nur bei Python!

Attribute mit self.<Name>

Eine Methode

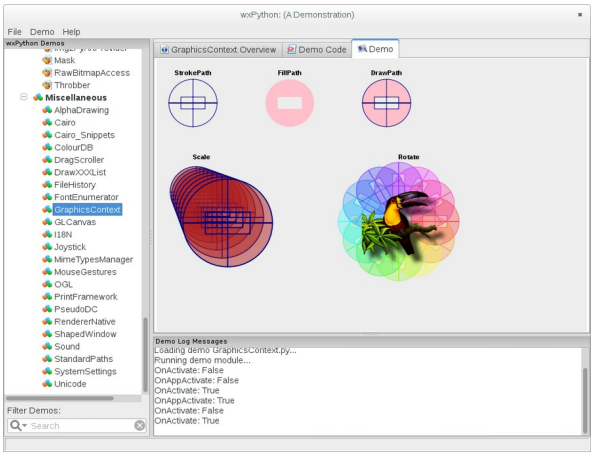
Einrückungstiefen definieren die Programmstruktur

```
    def GibSichtbar(self):
        """Get-Methode fuer die Sichtbarkeit"""
        return self.s

    def BewegeHorizontal(self, weite):
        """Veraendernde Methode fuer die x-Position"""
        self.Verberge()
        self.x += weite
        self.Zeige()

    def BewegeVertikal(self, weite):
        """Veraendernde Methode fuer die y-Position"""
        self.Verberge()
        self.y += weite
        self.Zeige()

    def Drehe(self, winkel):
        """Veraendernde Methode fuer die Orientierung [Winkel]"""
        self.Verberge()
        self.w += winkel
        self.Zeige()

    def Verberge(self):
        """Veraendernde Methode fuer die Sichtbarkeit mit Wert False"""
        self.s = False
        Zeichenflaeche.GibZeichenflaeche().Entferne(self)

    def Zeige(self):
        """Veraendernde Methode fuer die Sichtbarkeit mit Wert True"""
        self.s = True
        Zeichenflaeche.GibZeichenflaeche().Zeichne(self)



### --------------- Tisch -----------------------------------
from grafikfenster import *                                    1
from math import radians

class Tisch():
    """Klasse Tisch
    ermoeglicht das Zeichnen und Bearbeiten eines
    Tisch-Symbols fuer den Raumplaner"""

    def __init__(self,
                 xPos=60,
                 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
```

1 Ein erneuter Import ist innerhalb der selben Datei nicht notwendig.
Anders als bei Java ist es aber zulässig, mehrere Klassen innerhalb einer Datei zu definieren, allerdings ist es besser, die Definitionen in zwei getrennten Dateien abzulegen.

```
        self.t=tiefe
        self.w=winkel
        self.f=farbe
        self.s=sichtbar
        if sichtbar: self.Zeige()

    def GibFigur(self):
        """definiert und transformiert die zu zeichnende Figur"""
        gc = Zeichenflaeche.GibZeichenflaeche().GibGC()
        path = gc.CreatePath()

        path.AddRectangle(0, 0, self.b, self.t)

        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

    def GibFarbe(self):
        """Get-Methode fuer die Farbe"""
        return self.f

    def GibSichtbar(self):
        """Get-Methode fuer die Sichtbarkeit"""
        return self.s

    def BewegeHorizontal(self, weite):
        """Veraendernde Methode fuer die x-Position"""
        self.Verberge()
        self.x += weite
        self.Zeige()

    def BewegeVertikal(self, weite):
        """Veraendernde Methode fuer die y-Position"""
        self.Verberge()
        self.y += weite
        self.Zeige()

    def Drehe(self, winkel):
        """Veraendernde Methode fuer die Orientierung [Winkel]"""
        self.Verberge()
        self.w += winkel
        self.Zeige()

    def Verberge(self):
        """Veraendernde Methode fuer die Sichtbarkeit mit Wert False"""
        self.s = False
        Zeichenflaeche.GibZeichenflaeche().Entferne(self)

    def Zeige(self):
        """Veraendernde Methode fuer die Sichtbarkeit mit Wert True"""
        self.s = True
        Zeichenflaeche.GibZeichenflaeche().Zeichne(self)

### ------------------------------------------------
```

Demo-Code für die Grafik



Grundsätzlich kann der Programmcode für die Grafik aus dem Beispiel *"Graphics Context"* der *"wxPython docs, demos and tools"* übernommen werden. Er muss allerdings an einigen Stellen modifiziert werden, was hier aber nicht erläutert[1] werden soll.
Die wichtigsten Anmerkungen:

- Die Klasse muss eine Sammlungsstruktur, eine Liste bereitstellen, im Programmtext heißt sie **`objekte`**.
- In dieser Liste werden die darzustellenden Objekte gehalten, damit die Zeichenmethode darauf zugreifen kann.
- Damit Objekte in diese aufgenommen werden, muss es eine zuständige Methode geben, die hier **`self.Zeichne(objekt)`** heißt. Diese Methode wird vom einzigen Zeichenflaeche-Objekt bereitgestellt, das wir die (Klassen-) Methode Zeichenflaeche.GibZeichenflaeche() [2] erhalten. Die Methode zum Löschen der Darstellung des Symbols heißt **`self.Entferne(objekt)`** .

Die Anwendungsklasse

Ein Programm mit grafischer Oberfläche benötigt eine Anwendungsklasse. Sie dient einmal dazu, aus der read-eval-print-loop von IDLE in den Zustand Ereignis-gesteuerter Programmierung zu wechseln.

Außerdem muss das von ihr erzeugte Anwendungsobjekt **`app`** zur Darstellung der Grafikelemente das Frame-Objekt **`fenster`** mit dem Titel Raumplaner-Grafik mit seiner Inhaltsfläche **`panel`** initialisieren und anzeigen.

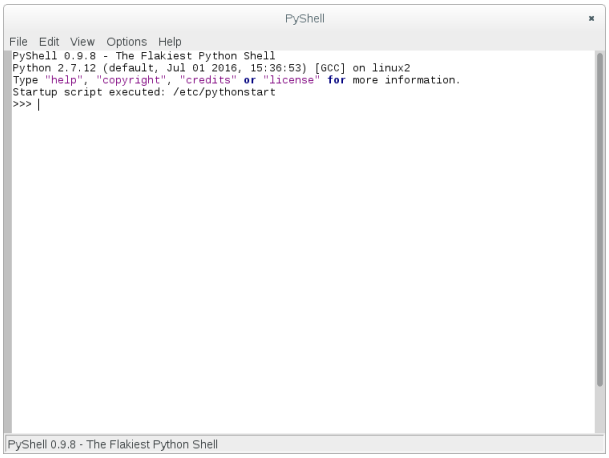
Von der Initialisierungsmethode wird im Anfangsprojekt zusätzlich eine Methode mit dem Namen ***`TestAnwendung`*** aufgerufen. Diese Methode zeichnet zunächst einen Tisch und einen Stuhl auf die Zeichenfläche.

```
### ---------- Die Anwendungsklasse -----------------------------------------
## bei getrennten Dateien die Importe einbauen:
from grafikfenster import *
from stuhl import *
from tisch import *

class RaumplanerApp(wx.App):
    """Testanwendung fuer Raumplaner-Grafik
    ShellFrame: Interaktion mit laufender Anwendung
    FillingFrame: Anzeige der Umgebung"""
```

1 Man könnte die Grafik-Klasse natürlich auch kapseln [s.u.]
2 Hier wird das Entwurfsmuster Singleton eingesetzt: Es gibt genau ein Zeichenflaeche-Objekt.

```
    def OnInit(self):
        self.fenster = GrafikFenster(None, "Raumplaner-Grafik")
        self.SetTopWindow(self.fenster)
        self.fenster.Show(True)
        self.fenster.panel.Refresh()
        self.fenster.ZeigeShellFrame()
        #self.fenster.ZeigeFillingFrame()
        self.TestAnwendung()
        return True

    def TestAnwendung(self):
        """Allein fuer die Testanwendung:"""
        global tisch, stuhl
        tisch=Tisch(70, 30, 120, 60, 0, 'red', True)
        stuhl=Stuhl(20, 40, 40, 40, 270, 'blue', True)

### ----------- ihr bedingter Aufruf ----------------------------------------
if __name__ == '__main__':[1]
    app = RaumplanerApp(redirect=False)
    app.MainLoop()
```

Interaktives Arbeiten mit dem einfachen Programm



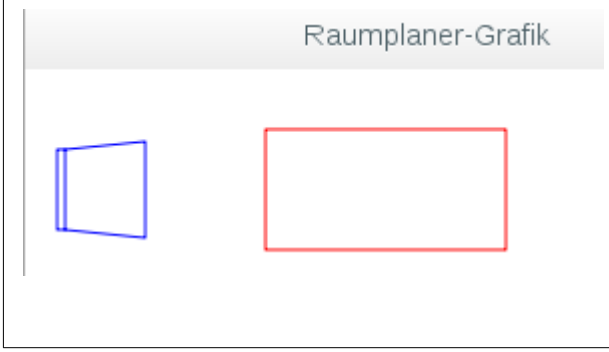
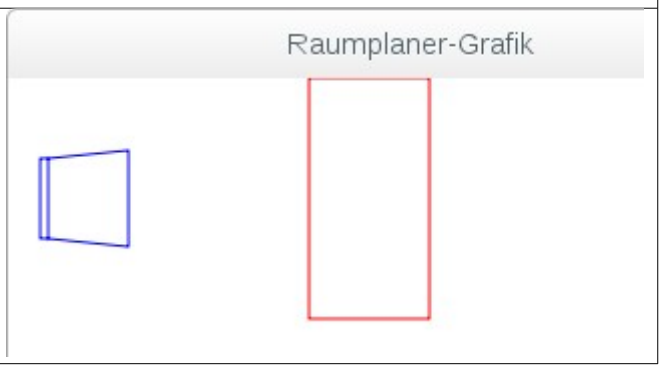
Ist die Zeile
`self.fenster.ZeigeShellFrame()`
nicht auskommentiert, wird die Anwendung mit einer Python-Shell umhüllt, mit der interaktiv auf die laufende Anwendung zugegriffen werden kann.
Die PyShell bietet Programmtext-Vervollständigung, so dass man nach dem Eingeben von **`app.`** alle Attribute und Methoden des Application-Objektes angezeigt bekommt. Man benötigt davon in der Regel die beiden von der Testanwendung zu globalen Variablen erklärten Objekte **`stuhl`** und **`tisch`**.

Eine Beispielanzeige:

```
>>> tisch
<__main__.Tisch instance at 0x1426d40>
```

Wir können auf diesem Wege die bestehenden Objekte manipulieren, auch neue erzeugen usw.

1 Das Einhüllen in die Abfrage führt dazu, dass diese Anwendung nur gestartet wird, wenn sie das Hauptprogramm ist. So können die Klassen Stuhl und Tisch in eigene Dateien ausgegliedert werden und dort ihre eigene Application-Klasse haben.

Die Zeile `tisch.BewegeHorizontal(50)` führt zu der erwarteten Veränderung.	Auch die Drehung `tisch.Drehe(90)` funktioniert wie erwartet.
	

Ein zweiter Stuhl

```
>>> stuhl2=Stuhl()
```

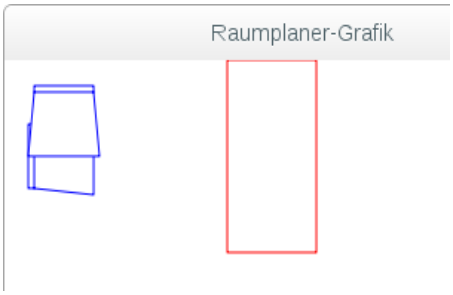
Fügen wir durch diese Eingabe im ShellFrame einen weiteren Stuhl hinzu, überrascht zunächst die Art des Aufrufs. Dem Konstruktor von Stuhl sind keine Parameter übergeben worden, ohne dass wir eine Fehlermeldung bekommen haben. Der Grund liegt in der Definition des Konstruktors, der vollständig mit vordefinierten Parametern versehen ist.

Will man andere Werte verwenden, wie es im Programmtext bei der Methode **`TestAnwendung`** geschehen ist, dann muss man sie in der richtigen Reihenfolge übergeben. Man kann allerdings auch Parameter auslassen, muss dann allerdings auch alle danach entsprechend <Parameter_Name>=<Parameter_Wert> übergeben.

Weiterhin überrascht sicher, dass der neue Stuhl nicht zu sehen ist. Sehen wir einmal im Programmtext nach. Der vordefinierte Parameter für die Sichtbarkeit ist **`False`**. Wir müssen dem Objekt zunächst noch mitteilen, dass es sichtbar sein soll. Dazu rufen wir die passende Methode auf:

```
>>> stuhl2.Zeige()
```

Nun können wir den Stuhl sehen[1]. Seine Position und Orientierung sind allerdings unbefriedigend.
Das kann uns allerdings nicht wirklich überraschen, da wir ja keine besonderen Werte definiert haben.



stuhl2 erzeugt und Zeige() aufgerufen

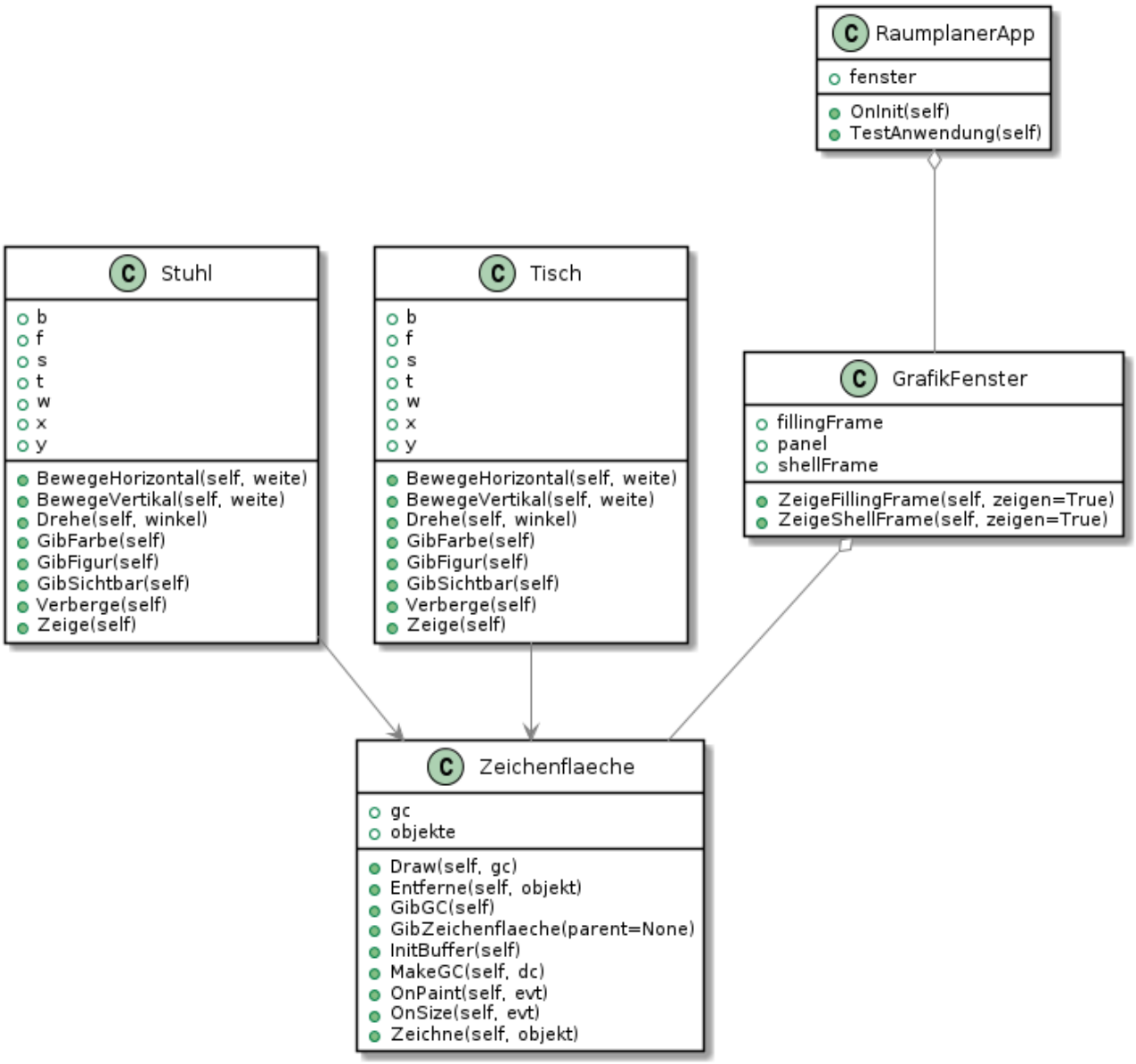
Noch keine gute Lösung

Die hier vorgestellte Lösung ist bewusst keine "gute Lösung". Sie stellt allein ein Anfangsprojekt dar, das "funktioniert" und die Schülerinnen und Schüler zu Erweiterungen motivieren soll. Beispielsweise führt ein Test der Eingabe von >>> stuhl2.f= 'red' (hoffentlich) auf viele weitergehende Fragen.

1 Wenn wir `stuhl2` gleich hätten sehen wollen, hätten wir ihn mit `Stuhl(sichtbar=True)` erzeugen können.

Das Klassendiagramm des Anfangsprojektes

Das abgebildete Klassendiagramm ist mit ***pyplantuml*** erstellt.
Es zeigt in den Klassen die Attribute und Methoden und die Beziehungen zwischen den Klassen. Auf diese Begriffe und ihre Bedeutung wird im Lauf der Bearbeitung des Projekts eingegangen.



Anhang: Beispiel für einen mit DIA erstellten Pythoncode

```
class Zeichenflaeche :
      '''Klasse fuer die Darstellung der Zeichenobjekte'''
      def __init__(self) :
            self.objekte = None # list
            pass
      def GibZeichenflaeche (self) :
            # returns Zeichenflaeche
            pass
      def Zeichne (self, neuesObjekt) :
            # returns
            pass
      def Entferne (self, dasObjekt) :
            # returns
            pass
class Grafikfenster :
      '''Frame fuer die Zeichenflaeche'''
      def __init__(self) :
            self.panel = None # Zeichenflaeche
            pass
      def ZeigeShellFrame (self, zeigen) :
            # returns
            pass
      def ZeigeFillingFrame (self, zeigen) :
            # returns
            pass
class Tisch :
      '''Stellt ein Tischsymbol auf der Zeichenflaeche dar'''
      def __init__(self) :
            self.x = 10.0 # float
            self.y = 10.0 # float
            self.b = 60.0 # float
            self.t = 120.0 # float
            self.w = 0.0 # float
            self.f = red # String
            self.s = True # Boolean
            pass
      def GibFigur (self) :
            # returns GraphicsPath
            pass

      # Get-Methoden wegegelassen

      def BewegeHorizontal (self, weite) :
            # returns
            pass
      def BewegeVertikal (self, weite) :
            # returns
            pass
      def Drehe (self, winkel) :
            # returns
            pass
      def Verberge (self) :
            # returns
            pass
      def Zeige (self) :
            # returns
            pass
```

```
class RaumplanerApp :
      '''Anwendungsklasse fuer das Projekt'''
      def __init__(self) :
            self.fenster = None # Grafikfenster
            pass
      def TestAnwendung (self) :
            # returns
            pass
class Stuhl :
      '''Stellt ein Stuhlsymbol auf der Zeichenflaeche dar'''
      def __init__(self) :
            self.x = 10.0 # float
            self.y = 10.0 # float
            self.b = 40.0 # float
            self.t = 40.0 # float
            self.w = 0.0 # float
            self.f = red # String
            self.s = True # Boolean
            pass
      def GibFigur (self) :
            # returns GraphicsPath
            pass

      # Get-Methoden wegegelassen

      def BewegeHorizontal (self, weite) :
            # returns
            pass
      def BewegeVertikal (self, weite) :
            # returns
            pass
      def Drehe (self, winkel) :
            # returns
            pass
      def Verberge (self) :
            # returns
            pass
      def Zeige (self) :
            # returns
            pass
```