GUI

Das Ziel unserer bisherigen Arbeit mit Python[1] ist gewesen, eine Oberfläche zu bieten, bei der man ohne eine GUI auskommt. Beim Erstellen einer GUI (graphical user interface), also einer grafischen Benutzerschnittstelle zu unserer Anwendung, wollen wir jetzt insbesondere die Unterstützung der **wxPython Demo** in Anspruch nehmen.
Wenn wir das Demo-tool starten, bekommen wir eine lange Liste von Beispiel-Anwendungen angezeigt. U.a. finden wir darin das Beispiel „GraphicsPath“, mit dem wir ständig gearbeitet haben, ohne es uns einmal genauer anzusehen. Das sollten wir einmal tun und dabei auch gleich versuchen, diese Anwendung aus dem Demo-tool in eine eigene Anwendung zu extrahieren. Dabei können wir auf unser bisheriges Projekt Bezug nehmen, sonst vielleicht lieber zunächst ein einfacheres Beispiel wählen, wie einen Frame.

Frame [im Abschnitt Frames und Dialogs]

Klickt man dies Beispiel an, bekommt man im Demo-Code im oberen Abschnitt den Teil, den wir benötigen, nämlich einmal den Import von wx und die Definition der Klasse MyFrame

```
import  wx

class MyFrame(wx.Frame):
    def __init__(
            self, parent, ID, title, pos=wx.DefaultPosition,
            size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE
            ):

        wx.Frame.__init__(self, parent, ID, title, pos, size, style)
        panel = wx.Panel(self, -1)

        button = wx.Button(panel, 1003, "Close Me")
        button.SetPosition((15, 15))
        self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)


    def OnCloseMe(self, event):
        self.Close(True)

    def OnCloseWindow(self, event):
        self.Destroy()
```

Allerdings fehlt hier noch die Anwendung, die diesen Frame startet. Eine Lösung dafür ist:

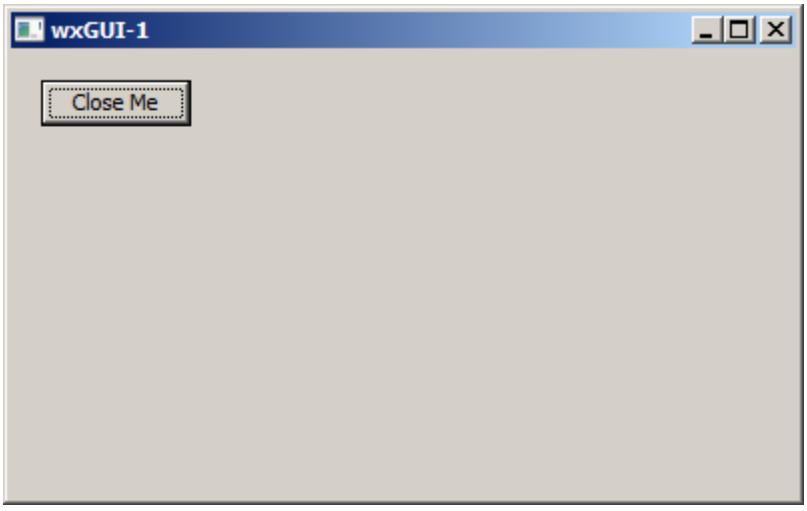
```
class MyApp(wx.App):
    def OnInit(self):
        self.fenster = MyFrame(None, -1, "wxGUI-1")
        self.SetTopWindow(self.fenster)
        self.fenster.Show(True)
        return True
```

1 Und das war das Ziel bei der Entwicklung von BlueJ. Das Erstellen einer GUI zu einer Java- Anwendung, fällt mit dem Javaeditor (http://lernen.bildung.hessen.de/informatik/javaeditor/index.htm) sehr viel leichter als mit BlueJ.

Von dieser Anwendungsklasse müssen wir nun noch eine Instanz erzeugen und von dieser Instanz die Methode MainLoop() aufrufen.

```
if __name__ == '__main__':
    app = MyApp(redirect=False)
    # Parameterwert True, wenn Ausgaben in der Standard E/A angezeigt
werden sollen
    app.MainLoop()
```

Nun kann das Ganze gestartet und – wie es der Button vorgibt – auch beendet werden:



Starten ohne eigene Anwendungsklasse

Das Starten gelingt auch ohne eine eigene Anwendungsklasse, indem man direkt ein App-Objekt erzeugt und benutzt. Diese Kurzform sieht dann so aus:

```
if __name__ == '__main__':
    app = wx.App()
    fenster = MyFrame(None,-1,title="wxGUI-1")
    fenster.Show()
    app.MainLoop()
```

Erläuterung des Programmtextes

- Zunächst muss natürlich das Paket wx importiert werden

```
import  wx
```

- Die Klasse erbt von wxFrame[1]

```
class MyFrame(wx.Frame):
    def __init__(
            self, parent, ID, title, pos=wx.DefaultPosition,
            size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE
            ):
```

- Dem Konstruktor müssen die Parameter
  - **`parent`**, also ggf. ein Fensterobjekt, von dem dieses abhängt
    [wenn es das nicht gibt, verwendet man den Parameter **`None`**],

1 Die Namen -auch in der Dokumentation von wxPython- beginnen alle mit wx und gehen ohne Punkt weiter. Wird auf eine Klasse in dem Paket zugegriffen, benötigt man den Namen wx für das Paket, den Punkt und nun den Namen ohne wx. Das ist leicht verwirrend, deshalb sollte man darauf hinweisen.

- eine **ID**, die man mit dem Wert -1 besetzen kann, dann wählt das System selbst eine und
- **titel**, also der in der Fensterzeile anzugebende Text, übergeben werden.

Alle anderen Parameter sind mit Standardwerten besetzt, können also übergeben werden, müssen aber nicht. Mit den Parametern **pos=(20,20), size=(200,100)** kann man beispielsweise ein kleineres (200x100 Pixel großes) Fenster erzeugen, das an der Position (20,20) mit der linken oberen Ecke auf dem Bildschirm dargestellt wird.

```
wx.Frame.__init__(self, parent, ID, title, pos, size, style)
```

- ruft den Konstruktor der vererbenden Klasse auf

```
panel = wx.Panel(self, -1)
```

- erzeugt das Objekt für die Inhaltsfläche

```
button = wx.Button(panel, 1003, "Close Me")
```

- erzeugt den Button; der zweite Parameter ist die ID, auch hier wäre -1 zulässig , der letzte seine Beschriftung (label).

```
button.SetPosition((15, 15))
```

- setzt ihn an die übergebene Position[1] in der Inhaltsfläche

```
self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button)
```

- erzeugt die Bindung zwischen dem Ereignis **wx.EVT_BUTTON** und der weiter unten definierten Methode **OnCloseWindow(...)** zum Schließen des Fensters. Das explizite Binden an diesen Button durch den dritten Parameter wäre hier nicht zwingend notwendig, bei mehreren Buttons allerdings schon. Alternativ geht auch button.Bind(wx.EVT_BUTTON, self.OnCloseMe), was im Hintergrund eine engere Bndung an den Button auslöst. Der Unterschied zur o.a. Variante sollte für unsere Zwecke allerdings nicht relevant sein.

```
self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
```

- Nun ist der Konstruktor beendet und es kommen die Methoden zur Ereignisbehandlung; sie bekommen vom System den Parameter **event** übergeben, über den man auf das Ereignisobjekt zugreifen kann, um Informationen zu bekommen oder Methoden aufzurufen. In diesem Beispiel geschieht das allerdings nicht.

  Will man wissen, was das event-Objekt alles an Attributen und Methoden hat, kann man zu Testzwecken in die Methode den Aufruf der Standardfunktion dir(...) einbauen, also print dir(event) und erfährt dann z.B., dass es eine Methode **GetId()** gibt, die in diesem Beispiel mit dem Aufruf **print event.GetId()** natürlich den Wert 1003 liefert.

```
def OnCloseMe(self, event):
    self.Close(True)
```

- Die erste Ereignismethode behandelt das Anklicken des Buttons, die zweite das Ereignis des Schließens des Fensters, das vermutlich durch das Anklicken des kleinen Kreuzchens in der Fensterleiste ausgelöst wurde.

```
def OnCloseWindow(self, event):
    self.Destroy()
```

1 Übergabe der Position als Tupel

Was jeder wxFrame schon kann

Ein großer Teil der benötigten Eigenschaften und Methoden wird schon von wxFrame[1] geerbt. In unserem Beispiel bekommt das Panel-Objekt keinen Layout-Manager, das sind von der Klasse wxSizer abgeleitete Klassen, also beispielsweise die Klasse wxBoxSizer. Die Layout-Manager ermöglichen standardisierte Layouts der Oberfläche, in unserem Beispiel ist das Layout daher absolut und statisch. Mit Grid-Layouts werden wir uns in passendem Zusammenhang noch beschäftigen.
Eine Komponentengruppe, die nicht dem Panel, also der Inhaltsfläche des Fensters, zugeordnet ist und dennoch Ereignis-gesteuert ist, sind die Menüs. Ein Menü tritt hier noch nicht auf. Wie man damit umgeht, werden wir uns also auch noch ansehen.

REPL, Ereignissteuerung und Ereignisbehandlung

Der Schritt von Programmen, die allein auf der Kommandozeile ausgeführt werden, zu Programmen, die Ereignis-gesteuert arbeiten, ist ein ganz Wesentlicher. Reine Kommandozeilenprogramme arbeiten nach dem Konzept der read-eval-print-loop[2]. Der Benutzer gibt in die Kommandozeile eine Anweisung ein, die vom Programm verarbeitet wird. Der Benutzer wartet[3] währenddessen auf das Ergebnis, also irgendeine Ausgabe.

Bei Ereignis-gesteuerten Programmen benötigen wir eine Anwendung, die gestartet wird, die in der Regel einige Aktionen ausführt, beispielsweise wie im oben betrachteten einfachen Beispiel ein Fenster aufbaut und dann auf das Auftreten von Ereignissen wartet. In der Zwischenzeit kann der Computer andere Programme bearbeiten. Alle am System angemeldeten Threads werden vom Betriebssystem regelmäßig abgefragt, bekommen also Rechenzeit zur Verfügung, um ihre Aufgabe zu erledigen. Dass dies nicht immer reibungsfrei abläuft, hat jeder von uns schon einmal erlebt, soll hier aber nicht weiter betrachtet werden.
Beispiele von Ereignissen sind ein **`wxEVT_BUTTON`** bei einem angeklickten Button und das **`wxEVT_CLOSE`**, das beim Schließen eines Fensters ausgelöst wird.
Objekte, die auf Ereignisse reagieren sollen, sind dafür verantwortlich, selbst oder mit Hilfe anderer die Behandlung auszulösen. Was beim Auftreten eines Ereignisses geschieht, muss in den Methoden zur Ereignisbehandlung beschrieben werden. Bei Python wird die Verknüpfung zwischen Ereignis und Ereignisbehandlungsmethode über die Methode Bind(...) hergestellt. Die Klasse Window ist abgeleitet[4] von der Klasse EventHandler, welche die notwendige Funktionalität bereitstellt.

Funktionalität fehlt noch → Raumplaner einbauen

Die Funktionalität fehlt noch. Das müsste aber bedeuten, dass wir eine sinnvolle Minimal-Anwendung definieren sollten. Wir binden daher nun unser Raumplanerprojekt ein. Dazu holen wir uns die Dateien eines Raumplanerprojektes mit unserer einfachen Gui-Datei in ein eigenes Verzeichnis.

---

1 Beachten Sie den Unterschied der Schreibweise beim Namen der Klasse in der Klassendefinition, hier wxFrame und der Verwendung im Programm wx.Frame, also der Klasse Frame aus dem modul wx.
2 Im deutschen Sprachraum wird viel der Begriff "EVA-Prinzip" verwendet, also Eingabe Verarbeitung Ausgabe.
3 Dieses Warten musste man bei alten Versionen von MSWindows sehr ernst nehmen. Nach der Aufforderung an einen Drucker, etwas auszudurcken, war das System erst wieder benutzbar, wenn der Drucker fertig war. Threads, also [scheinbar] nebenläufige Prozesse waren da noch nicht vorgesehen.
4 U.a. von der Klasse: Mehrfachvererbung!

Minimalanforderung an die Gui zum Raumplaner

Wir wollen von einem Button-Klick ausgelöst ein Stuhl-Objekt erzeugen, das auf der Zeichenfläche dargestellt wird. Dazu benötigen wir einen weiteren Button. Im Programmtext der Gui-Klasse sollten wir daher den Namen des bisherigen Buttons durch einen sinnvollen Namen ändern, beispielsweise nennen wir ihn jetzt **`endeButton`** und beschriften ihn mit Beenden. Die Definition der Position nehmen wir mit in den Konstruktoraufruf auf, so dass er jetzt lautet:

```
endeButton = wx.Button(panel, 1003, "Beenden", pos=((220, 110)))
```

Entsprechend schreiben wir die Definition für den Button, der zum Erzeugen des Stuhl-Objektes angeklickt werden soll.

```
stuhlButton = wx.Button(panel, 1004, "neuer Stuhl", pos=((220,10)))
```

Nun benötigen wir eine Ereignisbehandlungsmethode, an die wir das Ereignis Mausklick auf den **`stuhlButton`** binden wollen. Die beiden Bindungen[1] lauten daher jetzt:

```
self.Bind(wx.EVT_BUTTON, self.OnCloseMe, endeButton)
self.Bind(wx.EVT_BUTTON, self.OnStuhl, stuhlButton)
```

Die Methode OnStuhl muss die Botschaft (message), dass ein Stuhl-Objekt erzeugt werden soll, an irgendein dazu fähiges Objekt schicken können[2]. Bisher ist dafür das Objekt app, also das durch die Klasse **`MyApp`** definierte Anwendungs-Objekt der Raumplaner-Anwendung zuständig. Um Verwechslungen auszuschließen benennen wir diese Klasse sinnvoll um zu **`RaumplanerApp`**.
Ohne weiter über die angemessene Modellierung nachzudenken – das müssen wir allerdings noch nachholen – entscheiden wir uns, an das app-Objekt die notwendige message zu schicken. Dazu muss die Gui-Klasse allerdings das app-Objekt kennen. Es sollte der Gui beim Erzeugen übergeben werden, so dass wir den Konstruktor von Gui um den Parameter app ergänzen und ihn in ein Attribut schreiben. Der Kopf des Konstruktors von Gui sieht nun so aus:

```
def __init__(
        self, parent, ID, title, pos=wx.DefaultPosition,
        size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE, app=None
        ):
    . . .
    self.app = app
```

Nun können wir die Ereignisbehandlungsmethode für den Klick auf den **`stuhlButton`** schreiben:

```
def OnStuhl(self, event):
    self.app.StuhlErzeugen()
```

Die entsprechende Methode muss in die Klasse **`RaumplanerApp`** aufgenommen werden.

---

1 Alternativ – ohne hier auf die Unterschiede einzugehen:
`self.endeButton.Bind(wx.EVT_BUTTON, self.OnCloseMe)`

2 Alternativ kann man in einer Anfangsversion zunächst einfach den Kontruktor von Stuhl in dieser Methode aufrufen. Dann ist der Parameter app im Kontruktor unnötig.

```
def StuhlErzeugen(self):
    self.stuhl=Stuhl(50,50, farbe='blue', sichtbar=True)
```

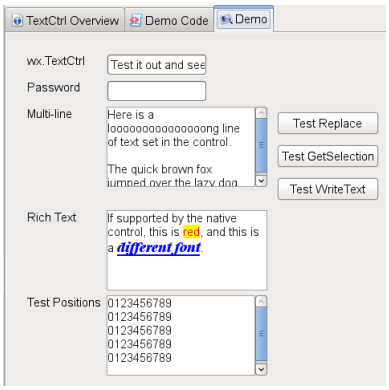
Das ist aber nicht die einzige notwendige Änderung.

Änderungen in der Klasse RaumplanerApp

Den Aufruf von TestAnwendung im Konstruktor der Klasse **`RaumplanerApp`** kommentieren wir aus, da wir die Testanwendung nicht mehr brauchen.
Da unsere Gui nun das Hauptfenster sein soll, müssen wir dies im Konstruktor korrigieren. Die gesamte **`OnInit(...)`**-Methode von **`RaumplanerApp`** sieht nun so aus:

```
def OnInit(self):
    self.gui=Gui(None, -1, 'Gui', pos=((200,50)), size=(400,300), app=self)
    self.SetTopWindow(self.gui)
    self.fenster = GrafikFenster(self.gui, "Raumplaner-Grafik")
    self.fenster.Show(True)
    self.fenster.panel.Refresh()
    self.fenster.ZeigeShellFrame()
    #self.fenster.ZeigeFillingFrame()
    self.gui.Show(True)
    return True
```

Der Aufruf der Show(...)-Methode von gui steht sinnvollerweise am Ende, damit es als letztes und damit ganz vorn angezeigt wird.

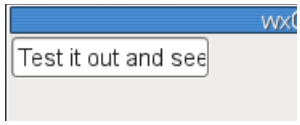


Einbau eines Textfeldes [Eingabefeld, TextCtrl]

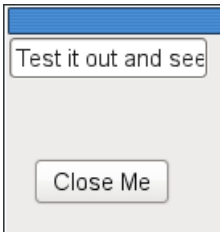
Entsprechend lässt sich auch ein Eingabefeld [Textfeld, bei wxPython wx.TextCtrl] in die einfache Anwendung einfügen.
Wir holen das Beispiel aus dem Demo-Tool und zwar unter ***Core Windows/Controls***. Es zeigt viele verschiedene Varianten von Textfeldern[1].
Wir beschäftigen uns hier nur mit der einfachsten, der ersten Variante. Im Demo-Beispiel finden wir den Programmtext

```
t1 = wx.TextCtrl(self, -1, "Test it out and see", size=(125, -1))
wx.CallAfter(t1.SetInsertionPoint, 0)
self.tc1 = t1
```

der komplizierter ist, als es sein müsste. Um zu sehen, was wir wirklich brauchen, bauen wir Zeile für Zeile in unser einfaches erstes Gui-Projekt ein. Der erste Test mit der ersten Zeile verläuft zunächst unbefriedigend, wie das Bild rechts zeigt.



Der Grund ist die Positionierung. Wenn wir bei statischen Layouts bleiben, wäre die einfachste Variante den Button nach unten zu verschieben. Wir ändern die Positionierung aus dem o.a. Beispiel durch



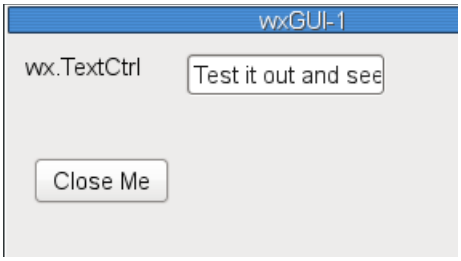

```
button.SetPosition((15, 75))
```

und erwarten nun beide Elemente auf der Oberfläche zu sehen, was aber

1 Daneben finden wir auch Beschriftungen [Java: JLabel] auf der Oberfläche. Sie heißen bei wxPython StaticText.

leider nicht der Fall ist.
Das Problem ist leider etwas komplizierter und liegt daran, dass in der Definition von t1 der erste Parameter self ist, also das Frame-Objekt bezeichnet. Wir wollen das Textfeld aber auf das Panel-Objekt zeichnen und müssen daher den ersten Parameter auf panel setzen. Nun ist das Ergebnis befriedigender. Die anderen beiden Zeilen scheinen nicht notwendig zu sein[1], daher entfernen wir sie nach einem Versuch wieder.

Beschriftung, Label, StaticText



Was bei Java ein **`JLabel`** ist, also ein Text, bei dem nicht das Ziel ist, ihn bearbeiten zu können, heißt bei wxPython **`StaticText`**. Wenn wir auch ihn aus dem Demo-Beispiel holen, sieht das Bild wieder zunächst unbefriedigend aus, da wir nun zwingend positionieren müssen. Mit den Ergänzungen

```
        l1 = wx.StaticText(panel, -1, "wx.TextCtrl", pos=((10,10)))
        t1 = wx.TextCtrl(panel, -1, "Test it out and see",
                         size=(125, -1), pos=((110,10)))
```

sieht das Fenster dann wie rechts dargestellt aus.

Einbau in die einfache Raumplaner-Anwendung

Auch diese beiden Elemente fügen wir unserer einfachen Raumplaner-Anwendung hinzu. Das Stuhl-Objekt soll beim Erzeugen eine in das TextCtrl-Objekt eingegebene Farbe haben.
Dazu muss die Methode in der Klasse **`RaumplanerApp`** so verändert werden, dass sie eine Farbe übergeben bekommen kann.

```
    def StuhlErzeugen(self, farbe='black'):
        self.stuhl=Stuhl(50,50, farbe=farbe, sichtbar=True)
```

Um den eingegebenen Farbnamen auslesen zu können, muss die Ereignisbehandlungsmethode auf das TextCtrl-Objekt zugreifen können. Das geschieht mit der Methode **`GetValue()`**. Die entsprechende Methode zum Setzen heißt natürlich **`SetValue(...)`**.
Die Gui-Klasse lautet nun:

```
        endeButton = wx.Button(panel, 1003, "Beenden")
        endeButton.SetPosition((220, 110))
        stuhlButton = wx.Button(panel, 1004, "neuer Stuhl", pos=((220,10)))
        self.Bind(wx.EVT_BUTTON, self.OnCloseMe, endeButton)
        self.Bind(wx.EVT_BUTTON, self.OnStuhl, stuhlButton)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)

        l1 = wx.StaticText(panel, -1, "Farbe", pos=((10,10)))
        self.t1 = wx.TextCtrl(panel, -1,
                              "Farbname", size=(125, -1), pos=((80,10)))

    def OnStuhl(self, event):
        self.app.StuhlErzeugen(self.t1.GetValue())
```

1 Eine Einfügeposition zu definieren, scheint nicht sinnvoll zu sein. Auch die Zuweisung zu einem Attribut des Frame-Objektes ist noch nicht sinnvoll.

t1 wird zum Attribut

In diesem Text fällt auf, dass das TextCtrl-Objekt **`t1`** nun zu einem Attribut gemacht wurde durch **`self.t1`**. Das ist notwendig, da wir von außerhalb des Konstruktors, nämlich von der Ereignisbehandlungsmethode aus, darauf zugreifen wollen. **`l1`** dagegen kann weiterhin eine lokale Variable des Konstruktors bleiben.

Aufgabe:

Ändern Sie das Projekt so ab, dass in das Textfeld die xPosition des Stuhl-Objektes eingegeben werden kann.

Andere Anforderungen

Grundlegende Eigenschaft eines TextCtrl-Objektes ist der Textinhalt des Feldes, den man mit den Methoden **`SetValue(...)`** setzen und mit **`GetValue()`** lesen kann. Der Rückgabewert der letzten Methode ist ein Stringobjekt, das man nun verarbeiten kann. Er ist zunächst selbst dann ein Stringobjekt, wenn der Benutzer Zahlen in das Textfeld eingegeben hat. Da der Benutzer diese Zahlen in der Regel nicht als Text verarbeitet haben möchte, muss eine Möglichkeit geben, diese Zeichenfolgen mit dem Sinn von Zahlen in die entsprechende Zahl zu verwandeln. Da hier ganzzahlige Werte eingegeben werden sollen, verwandeln wir sie mit der Funktion int in eine solche. Allerdings können wir ja nicht sicher sein, ob der Benutzer nicht doch etwas anderes eingegeben hat als eine natürliche Zahl und sei es vielleicht "hundert". Diese Fehlermöglichkeit sollten wir abfangen. Daher wird die Methode **`OnStuhl(..)`** geändert:

```
def OnStuhl(self, event):
    eingabe = self.t1.GetValue()
    try:
        self.app.StuhlErzeugen(abs(int(eingabe)))
    except ValueError as e:
        print str(e)
```

Welcher Fehler durch eine falsche Eingabe ausgelöst wird, kann man in der Shell z.B. mit int('elf') austesten und erfährt, dass ein **`ValueError`** abzufangen ist. Die Ausgabe dieses Fehlers in einer Konsolenmeldung ist keine endgültige Lösung. wxPython stellt dafür angemessene Dialoge bereit.
Die Methode **`StuhlErzeugen(...)`** in der Anwendungsklasse muss ebenfalls modifiziert werden.

```
def StuhlErzeugen(self, xPosition=50):
    self.stuhl=Stuhl(xPosition,50, sichtbar=True)
```