Kapselung

Die Einführungsaufgabe zu diesem Abschnitt zeigt, dass es möglich ist, auf das Attribut Farbe (im Projekt kurz **`self.f`**) von außen nicht nur lesend, sondern auch schreibend zuzugreifen. Die Anweisung

```
>>> zweiterStuhl.f = "blue"
```

ändert zwar den Attributwert, führt aber nicht gleichzeitig zu einer Änderung der Darstellung. Wie sollte das auch sein – das für die Darstellung zuständige Objekt weiß ja nichts von der Veränderung! Erst nach einem Aufruf von

```
>>> zweiterStuhl.Zeige()
```

ändert sich die Darstellung zu blauer Farbe. Unser Grafiksystem ist dadurch ohne diese zusätzliche Anweisung in einen inkonsistenten Zustand übergegangen: Daten und Darstellung stimmen nicht mehr überein.
Der Grund für dieses Problem ist aber keine fehlerhafte Anwendung des Benutzers, sondern ein Fehler im Entwurf. Der Fehler ist entstanden, weil unsere Klasse bisher keine Methode `AendereFarbe(neueFarbe)` bereitstellt.
Dies Problem zu lösen führt auf die Anwendung eines wichtigen Konzeptes Objekt-orientierter Programmierung, auf die Kapselung von Daten.

Kapselung

Auf die Daten eines Objektes sollte nicht direkt, sondern nur über die vom Entwickler bereitgestellten Get- und Set-Methoden zugegriffen werden können.

Vergleicht man mit den anderen Methoden der Klassen, dann stellt man fest, dass diese zunächst das Objekt aus der Darstellung entfernen, erst dann das Attribut neu setzen und das Objekt nun mit den veränderten Werten neu zeichnen. So gehen wir nun auch bei der Methode AendereFarbe vor.

```
    def AendereFarbe(self, neueFarbe):
        self.Verberge()
        self.f = neueFarbe
        self.Zeige()
```

Nun führen die Anweisungen

```
>>> zweiterStuhl=Stuhl()
>>> zweiterStuhl.Zeige()
>>> zweiterStuhl.AendereFarbe('blue')
```

zum gewünschten Ergebnis.

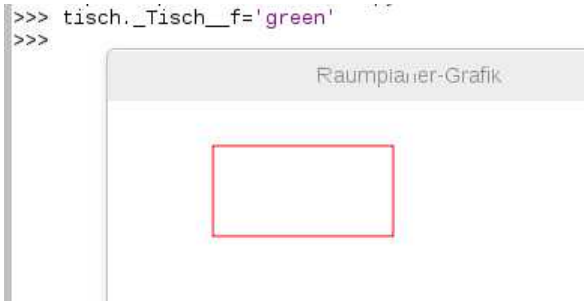
Realisieren der Kapselung in unserem Projekt

Wir haben soweit aber nur die fehlende Methode ergänzt, nicht aber die Kapselung der Daten der Möbelobjekte realisiert.
Eine wirksame Kapselung von Attributen ist laut Dokumentation von Python möglich, wenn diese im Namen durch zwei Unterstreichungsstriche[1] am Beginn als private Attribute gekennzeichnet werden. Wir ändern also unsere Attributdefinitionen in unserem Projekt in den Klassen **`Tisch`** und **`Stuhl`** entsprechend ab.

1 Mit einem einzelnen Unterstreichungsstrich gelten Attribute als geschützt nach Konvention. Das bedeutet, dass vom Programmierer bei einer sauberen Programmierung erwartet wird, dass er diese Möglichkeit nicht nutzt.

```
class Tisch:
    def __init__(self,
                 xPos=60,
                 yPos=10,
                 breite=120,
                 tiefe=60,
                 winkel=0,
                 farbe="black",
                 sichtbar=False):
        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):
        ...
    def Transformiere(self, path):
        ...

    def BewegeHorizontal(self, weite):
        self.Verberge()
        self.__x += weite
        self.Zeige()

    def BewegeVertikal(self, weite):
        self.Verberge()
        self.__y += weite
        self.Zeige()

    def Drehe(self, winkel):
        self.Verberge()
        self.__w += winkel
        self.Zeige()

    def AendereFarbe(self, neueFarbe):
        self.Verberge()
        self.__f = neueFarbe
        self.Zeige()

    def Verberge(self):
        self.__s = False
        Zeichenflaeche.GibZeichenflaeche().Entferne(self)

    def Zeige(self):
        self.__s = True
        Zeichenflaeche.GibZeichenflaeche().Zeichne(self)

    def GibFarbe(self):
        return self.__f

    ...
```

Wirksame Kapselung? Leider doch nicht!



Dennoch[1] ist es möglich, sogar schreibend auf "gekapselte" Attribute zuzugreifen. Man kann die Kapselung umgehen: Beginnt man im *ShellFrame* bei der Eingabe mit dem Objektnamen und einem Punkt und wartet kurz, dann werden zwar zunächst nur die öffentlichen Methoden und Attribute angezeigt, gibt man aber einen Unterstreichungsstrich ein, dann werden auch private angezeigt und man kann sie auswählen und normal auf sie zugreifen[2]. Im Bild ist mit **`tisch._Tisch__f = 'green'`** ein solcher schreibender Zugriff gezeigt und man erkennt, dass es nicht sinnvoll ist, die verändernde Methode AendereFarbe(…) auf diesem Wege zu umgehen, da Attributwert und dargestellte Farbe nicht übereinstimmen.

Get und Set [verändernde und sondierende Methoden]

Bisher haben wir die Set-Methoden *[verändernde Methoden]* umgesetzt – wenn wir sie auch nicht so genannt haben:
`BewegeHorizontal(...)` ist die schon vorhandene Set-Methode *[verändernde Methode]* für das Attribut x-Koordinate, **`BewegeVertikal(...)`** die für die y-Koordinate, für den Winkel **`Drehe(...)`** und **`AendereFarbe(…)`** die für Farbe[3].
Wir fügen zu allen Attributen Get-Methoden ein. Wozu sollten wir auch die Inhalte gegen Lesbarkeit schützen? Dazu erweitern wir die Klasse Moebel um die folgenden Methoden:

```
    def GibXPosition(self):
        return self.__x

    def GibYPosition(self):
        return self.__y

    def GibBreite(self):
        return self.__b

    def GibTiefe(self):
        return self.__t

    def GibWinkel(self):
        return self.__w

    def GibFarbe(self):
        return self.__f

    def GibSichtbarkeit(self):
        return self.__s
```

[4]

1 Wenn es Sie interessiert, recherchieren Sie gern zum Konzept der Kapselung bei Python. Es wird dort auch auf die Anwendung von @property verwiesen. Ich sehe darin hier aber keinen wesentlichen Zugewinn. Siehe dazu die Beispielprojekte, die allein beim Attribut sichtbar erkennbar vorteilhaft sind.
2 Dies lässt sich auch bei der Vererbung nutzen. Siehe dort.
3 Es fällt auf, dass wir keine Set-Methoden für die Breite und für die Tiefe der Moebel-Objekte haben. Sollte es zu den Anforderungen gehören, dass auch diese beiden Attribute verändert werden sollten, dann müsste man diese hinzufügen. Insbesondere eine Methode zur proportionalen Veränderung von Breite und Tiefe (→ Skalierung) gleichzeitig könnte sinnvoll sein.
4 GibFarbe(...) war vorher schon für den Zugriff durch die Klasse Zeichenflaeche eingebaut.

sichtbar als property

Das in der Anmerkung auf der vorderen Seite angegebene Beispiel ist hier auszugsweise angegeben.

```
def AendereFarbe(self, farbname):
    self.sichtbar=False
    self.__f = farbname
    self.sichtbar=True

@property
def sichtbar(self):
    return Zeichenflaeche.GibZeichenflaeche().IstSichtbar(self)

@sichtbar.setter
def sichtbar(self, wert):
    if wert: Zeichenflaeche.GibZeichenflaeche().Zeichne(self)
    else: Zeichenflaeche.GibZeichenflaeche().Entferne(self)
```

Extern kann beispielsweise mit **`tisch.farbe`** zugegriffen werden, so dass die Anweisung **`tisch._Tisch__sichtbar = True`** erfolgreich umgesetzt wird, da die angegebene verändernde Methode aufgerufen wird. In diesem Beispiel wird auch gezeigt, wie auf ein eigenes Attribut zur Speicherung verzichtet werden kann, da der Wert in der Klasse Zeichenflaeche bestimmt werden kann.

property kein Unterrichtsinhalt

Da die hier gezeigte Möglichkeit allein bei Python realisiert wird, sollte sie kein Unterrichtsinhalt sein. Das allgemeinere Konzept ist das der Sichtbarkeit.

Allgemeines zum Begriff Kapselung

Der Begriff Kapselung[1] wird nicht nur wie hier für die Objektorientierung beschrieben verwendet.
Im obigen Text geht es um die Kapselung in objektorientierten Systemen, bei der es darum geht, die Daten von Objekten vor unkontrollierten Zugriffen zu schützen und Zugriffe nur über die in der Klasse definierten öffentlichen Methoden zuzulassen. Erweitert auf die Programmstruktur heißt das, es wird allein eine Schnittstelle zur Verfügung gestellt, ohne dem Benutzer Informationen über die interne Realisierung zu geben[2].
Bei der strukturierten Programmierung beschreibt Kapselung die Bindung von Datenbereichen an Teilbereiche des Programms, in denen sie gelten und in denen auf sie zugegriffen werden kann und nur dort.
In der funktionalen Programmierung beispielsweise wird durch Aufruf einer Funktion ein lokaler Namensraum definiert, auf den von außen nicht zugegriffen werden kann.
Interessant wird dieses Konzept bei modernen Anwendungen von paralleler Programmierung.

1 encapsulation
2 Geheimnisprinzip; information hiding