Vererbung

Es ist sehr unbefriedigend, große Programmtextabschnitte durch Kopieren in andere Klassendefinitionen zu übertragen, ohne dass in weiten Teilen auch nur eine Zeile geändert wird. Absolut naheliegend ist, diese Abschnitte in einen eigenen Bereich auszugliedern, um dann jedesmal darauf zuzugreifen. In der OO ist das geeignete Mittel dafür eine neue, nur dazu dienende Klasse.

Vererbung bedeutet Abstraktion

In diesem Vorgehen steckt aber nicht nur ein Schreibkonstrukt, sondern eine Abstraktionsstufe. Wir suchen die ***gemeinsamen Attribute*** und ***Methoden*** aller Möbelklassen, ***abstrahieren*** dabei von einer konkreten Möbelklasse und definieren das Abstraktionsergebnis durch eine eigene Klasse **`Moebel`**. Diese stellt nun die allen anderen Klassen gemeinsamen Eigenschaften bereit und die von ihr erbenden [abgeleiteten] Klassen enthalten nur noch genau die Methoden, in denen sie sich unterscheiden.
Das ist in unserem Raumplanerprojekt ganz einfach. Die Klassen Stuhl, Tisch usw. enthalten nur noch den Konstruktor **`__init__(self, ...)`** und die Methode **`GibFigur()`**. Alle anderen Methoden werden in die Klasse **`Moebel`** ausgegliedert.

Vererbung bei der erbenden Klasse kennzeichnen

Es ergibt sich das Problem, dass unser Projekt nun zwar eine neue Klasse hat, die Klassen **`Stuhl`** und **`Python`** aber nichts davon wissen, dass die Möbelklassen ihre Attribute und Methoden von der Klasse **`Moebel`** erben sollen.
Das müssen wir in den Klassentext einarbeiten. Die Datei ***moebel.py*** muss als Modul in beispielsweise ***stuhl.py*** eingebunden werden. Da wir daraus nur die Klasse **`Moebel`** benötigen, schreiben wir vor dem Beginn der Klassendefition:

```
from moebel import Moebel
```

Der Kopf der Klassendefinition wird ergänzt um die Angabe der vererbenden Klasse (Oberklasse, Superclass) in Klammern hinter dem Namen der Klasse. Nun weiß Python, dass es es die dort angegebenen Attributen und Methoden einbinden soll.

Konstruktor der vererbenden Klasse aufrufen

Bei Python ist es ganz wichtig, im Konstruktor der erbenden Klasse (häufig zuerst) den Konstruktor der vererbenden Klasse aufzurufen. In unserem Beispiel muss also jede der konkreten Möbelklassen, also beispielsweise die Klasse **`Stuhl`**, einen deutlich veränderten, verkürzten Konstruktor bekommen. Er enthält allein die Zeile

```
Moebel.__init__(self, xPos, yPos, breite, tiefe, winkel, farbe, sichtbar)
```

Die Attributwerte werden also an den Konstruktor in der Klasse **`Moebel`** weiter gereicht und ihre Definition übernimmt nun die Klasse **`Moebel`**.

Moebel ist eine abstrakte Klasse[1]

Wenn wir die gemeinsamen Methoden in die neue Klasse Moebel ausgegliedert haben und die verkürzte Methode `gibFigur()` in der konkreten Möbelklasse verblieben ist, dann kennt die Klasse **`Moebel`** die Methode `gibFigur()` nicht. Es gibt aber Methoden in **`Moebel`**, die darauf zugreifen. Das muss zu einem Fehler führen, wenn wir versuchen ein Moebel-Objekt zu erzeugen.

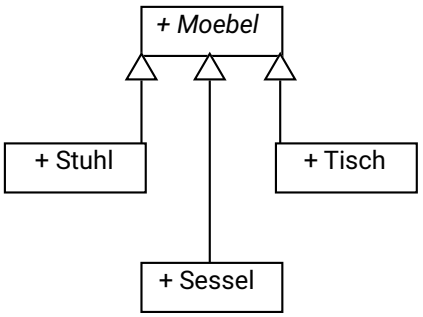


Abbildung 1: Klassendiagramm mit DIA erstellt

Wir können das Problem bei Python auf drei Arten lösen:

1. Wir lassen die Methode einfach weg und verlassen uns darauf, dass niemand versucht, die Klasse Moebel direkt zu instanziieren, was zu einem Fehler führen würde. Das erzeugen konkreter Möbelobjekte geht *[im Gegensatz zu Java]* bei Python gut, weil beim Erzeugen jedes konkreten Moebel-Objektes diese Methode von der Unterklasse zur Verfügung gestellt wird[2].
2. Wir erstellen eine eigene Methode `gibFigur()` in **`Moebel`**, die nichts tut. Das könnte prinzipiell eine leere Methode sein. Allerdings hat die Methode normalerweise eine Rückgabe, was durch `return None` erfüllt wäre.
   "Besser" ist die folgende Variante für den Programmtext in **`Moebel`**, die einen leeren Pfad zurückgibt:

```
def GibFigur(self):
    """Definiert den Pfad und transformiert ihn."""
    gc = Zeichenflaeche.GibZeichenflaeche().GibGC()
    path = gc.CreatePath()
    return path
```

   In jedem Fall wird die Methode `gibFigur()` von **`Moebel`** durch die jeweilige Methode der konkreten Möbelklasse überschrieben.

   Man könnte nun allerdings unsinnigerweise versuchen, eine Instanz der Klasse **`Moebel`** zu erzeugen. Das wäre deswegen sinnlos, da **`Moebel`** ein abstrakter Oberbegriff ist. *Es gibt kein Objekt **`Moebel`** an sich*. Die bessere Lösung ist also:
3. Wir definieren die Methode **`gibFigur()`** in Moebel als abstrakte Methode. In diesem Fall wird unsere Klasse aber notwendig eine abstrakte Klasse, also eine Klasse, von der keine Objekte erzeugt werden können.

   Das "Werkzeug" dafür ist das Modul abc [abstract base class], das wir zunächst importieren, dann die Klasse so kennzeichnen müssen und die Methode selbst dann als abstrakt kennzeichnen.

Die abstrakte Klasse **`Moebel`** sieht dann [gekürzt] so aus:

```
from abc import ABC,abstractmethod # abstrakte Klasse realisieren
### ------------------------------------------------
class Moebel(ABC): [3]
```

1 Klassendiagramm mit pyplantuml siehe am ende des Textes
2 Zum Testen: Methode GibFigur in der Klasse Moebel der nicht abstrakten Lösung auskommentieren.
3 Version für Python 2: **`class Moebel:`**
`__metaclass__ = ABCMeta`

```
    def __init__(self,
                 xPos=0,
                 yPos=0,
                 breite=200,
                 tiefe=80,
                 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()

    @abstractmethod
    def GibFigur(self):
        """Definiert den Pfad und transformiert ihn."""
        return NotImplemented

    . . .

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

    def Zeige(self):
        Zeichenflaeche.GibZeichenflaeche().Zeichne(self)
```

Der Versuch, ein Objekt Moebel zu erzeugen

```
moebel = Moebel(0,0,100,100,0,'black',True)
```

führt dann zu einer Fehlermeldung:

```
TypeError: Can't instantiate abstract class Moebel with abstract methods
GibFigur
```

Alle konkreten Möbelobjekte definieren die Methode **`GibFigur()`** selbst. Sie überschreiben die in **`Moebel`** definierte Methode, so dass bei ihnen kein Fehler auftritt.

Kapselung und Vererbung

Durch den Einsatz der gekapselten Attribute entstehen Probleme, wenn Methoden der erbenden Klassen auf diese zugreifen wollen. Das Problem ist in unserem Projekt leicht zu lösen, da nur die Methode **`GibFigur()`** in den erbenden Klassen auftaucht, die allein lesend auf die Attribute zugreift. Der Aufruf der passenden Get-Methoden löst das Problem.

Mehrfachvererbung

Nicht eingegangen ist in diesem Text auf die von Python angebotene Möglichkeit der

Mehrfachvererbung. Das hat zwei Gründe:

- Das bisherige Projekt bietet keine Begründung für Mehrfachvererbung und man sollte Inhalte nur dann besprechen, wenn es für sie vom Problem her einen Bedarf gibt.
- Java kennt keine Mehrfachvererbung. Diskussionen, ob man sie einsetzen sollte, grenzen oft an Auseinandersetzungen in Glaubensfragen. Java setzt dafür auf das Konzept der Interfaces [gibt es nicht vergleichbar in Python] und der Delegation und die ist in der Regel in Python oft auch die bessere Lösung.

Überschreiben und Polymorphie

Aus ähnlichen Gründen taucht der Begriff des Überschreibens nur knapp auf, wird nicht wirklich behandelt und Polymorphie ist noch gar nicht erwähnt. Das kann in einem sachlich gebotenen Kontext später nachgeholt werden, wenn auf das Entwurfsmuster Kompositum eingegangen wird. Sonst ist es wegen der in diesem Anwendungsproblem des Raumplaner-Projektes flachen Vererbung einer der wenigen Mängel, dass Polymorphie eigentlich keine Rolle spielt. Hier können ergänzende kleine Projekte[1] helfen, wenn man Polymorphie behandeln will.

1 Beispielsweise das Projekt mit den Zählerklassen

Vererbung begründen

Im oben angegebenen Text haben wir untersucht, wie Vererbung realisiert werden kann, haben als Grund für die Einführung der Klasse **`Moebel`** aber nur die ärgerliche Code-duplizierung angegeben. In dieser Codeduplizierung steckt das Problem der Redundanz und wir nutzen den Vorteil, die gemeinsamen Eigenschaften genau einmal beschreiben zu müssen. Das klingt vorrangig nach *"wir sparen uns Arbeit"*.

Redundanz vermeiden

Es steckt aber mehr hinter dem Vermeiden von Redundanz. Programmtext, der in gleicher Weise an mehreren Stellen auftaucht, muss bei Änderungen auch an mehreren Stellen geändert werden. Dies ist nicht wartungsfreundlich.

Es ist aber mehr als das: *Programmtext, der in gleicher Weise an mehreren Stellen auftaucht, muss bei Änderungen auch an mehreren Stellen geändert werden* – aber wird er das auch?

Die große Gefahr bei solchen identischen Programmtextabschnitten ist, dass bei einer Änderung zwar ein Abschnitt geändert wird, möglicherweise sogar viele dieser identischen Abschnitte, nicht aber alle. Das ist vielleicht geschehen, weil der Programmentwickler einen oder mehrere Abschnitte einfach vergessen hat oder weil er das später noch erledigen wollte oder vielleicht auch, weil er meinte, bei einem speziellen Beispiel käme das nicht so darauf an.

In jedem Fall ist das System inkonsistent, was möglicherweise zum Auftreten schlecht nachzuvollziehender Fehler führt und erst nach erheblicher Mühe und einem erneuten Überarbeiten korrigiert werden kann.

Wiederverwendung

Ein weiterer Aspekt der Vererbung ist die Möglichkeit von Wiederverwendung von vorhandenen Klassen in neuen Zusammenhängen. Will man beispielsweise für eine grafische Oberfläche einen Button programmieren, dann wird man nicht selbst den Programmtext schreiben, sondern von einer vorhandenen Klasse erben und nur die speziellen Eigenschaften hinzufügen, die man in dem speziellen Kontext braucht.

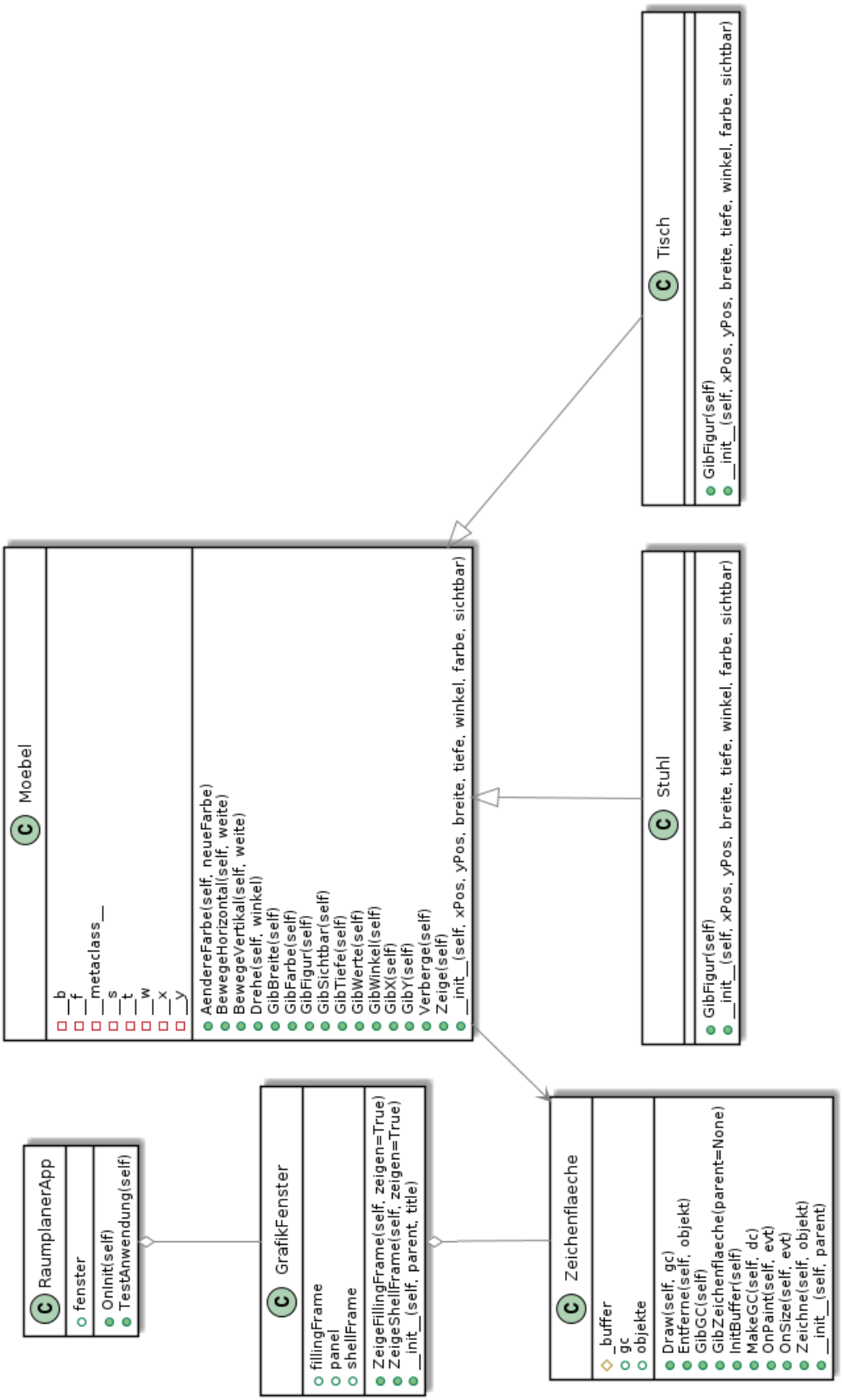
Ersetzbarkeit

Ein weiterer wichtiger Aspekt ist die Ersetzbarkeit von Objekten bei Zugriffen. Für eine Anwendung sollte es belanglos sein, welcher konkrete Typ [Subtyp, der von der Oberklasse erbt] vorliegt. Sie sollte so programmiert sein, dass allein die Kenntnis der Schnittstelle der Oberklasse notwendig ist, also der Methoden (und Attribute), die von der Oberklasse bereitgestellt werden. Dass die jeweils richtige zuständige Methode für das spezielle Objekt verwendet wird, dafür muss das Vererbungssystem sorgen.

Zusammenfassung

Vorteile von Vererbung

- Vermeiden von Codeduplizierung
- Wiederverwenden von vorhandenem Programmtext
- Einfache Wartung
- Erweiterbarkeit
- Ersetzbarkeit



Klassendiagramm mit pyplantuml erstellt