LU11c - Interface in Python

Schnittstellen spielen im Software Engineering eine wichtige Rolle. Wenn eine Anwendung wächst, werden Aktualisierungen und Änderungen an der Codebasis immer schwieriger zu verwalten. Hier erfahren Sie, wie Sie eine *Python-Schnittstelle* verwenden, um die Idee der Polymorphie auch „ausserhalb“ der Vererbungshierarchie anwenden zu können.

Grundsätzlich kann eine Schnittstelle als **Vertrag** verstanden werden. Sie definiert eine gewisse Funktionalität durch Methoden. Im Interface wird die Deklaration ohne Implementation festgehalten. (Anmerkung: man spricht oft auch von vollabstrakten Klassen, so z.B. bei C++)

Diese Funktionalität wird für die korrekte Funktion der Anwendung vorausgesetzt. Die implementierende Klasse nutzt die Deklaration der Schnittstelle, damit eine einheitliche Namensgebung gesichert ist und im Sinn der Polymorphie die Nutzung realisiert werden kann.

Beispiel: Objekte mit gleichartiger Funktionalität

Verschieden Objekte, die keine Beziehung zueinander aufweisen müssen, haben eine ähnliche Funktionalität, die sie nutzen.

Abb: Klassen mit sinngleicher Funktionalität

Obwohl die drei Klassen Mensch, Wasserflugzeug und Baum keine gemeinsame Basis (Oberklasse) haben, weisen alle die Fähigkeit `swim` auf. In UML wird diese Gegebenheit wie folgt dargestellt:

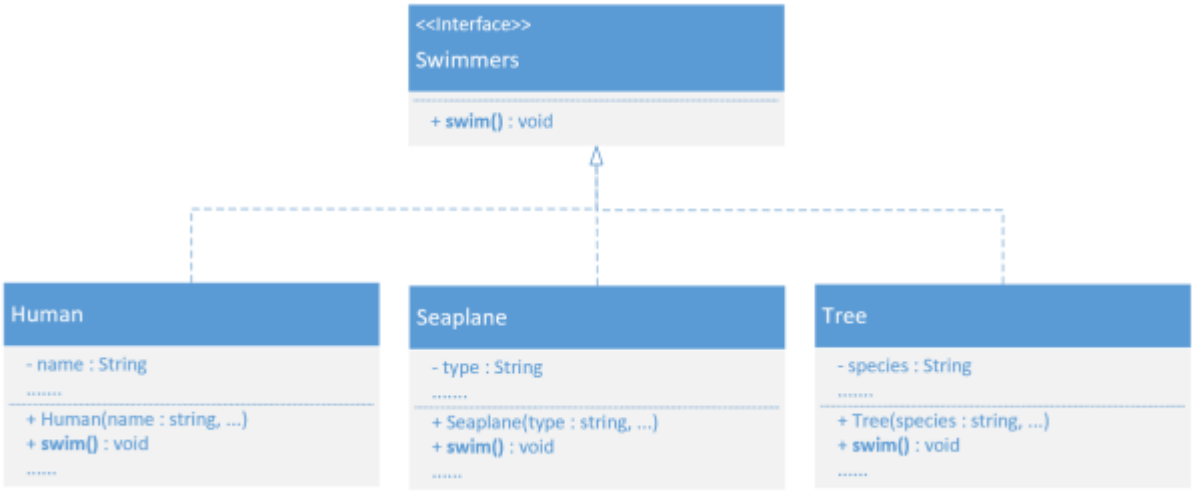

Abb: Schnittstellen in UML

Beachten Sie hier die Darstellung einer Schnittstellen-Beziehung als gestrichelte Linie (im Gegensatz zur Vererbung mit durchgezogener Linie).

Abb: UML-Symbole Vererbung und Schnittstelle

Das Beispiel verdeutlicht, was eine Schnittstelle - gegenüber einer Oberklasse - ausmacht. Im Code können entsprechende Objekte diese Funktionalität nutzen.

```
a_human_object = Human("Max", ....)
a_human_object.swim()
...
a_plane_object = Seaplan("Searay", ....)
a_plane_object.swim()
...
a_tree_object = Tree("Maple", ....)
a_tree_object.swim()
```

Werden diese verschiedenen Objekte in einer Liste eingetragen, so kann über alle Objekte iteriert und die entsprechende Methode `swim` ausgeführt werden.

Dieses Verhalten entspricht der Idee der Polymorphie. In Pyton wird das konkret mittels Duck-Typing umgesetzt.

Das Duck-Typing

Der Name *Duck-Typing* kommt von dem Satz:
Wenn es aussieht wie eine Ente und quakt wie eine Ente, ist es eine Ente.

Python kennt das Konstrukt des Interface nicht. Es wird ersetzt durch die Technik des *Duck-Typing*. Es stellt eine Typisierung dar, welche in dynamischen Sprachen verwendet wird. Nebst Python auch bei Perl, Ruby, PHP, Javascript usw., wo der Typ oder die Klasse eines Objekts weniger wichtig ist als die Methode, die es definiert. Mit Duck-Typing überprüfen wir Typen überhaupt nicht. Stattdessen prüfen wir, ob eine bestimmte Methode oder ein bestimmtes Attribut vorhanden ist.

Wozu Duck-Typing verwenden?

Dank *Duck-Typing* kann, ähnlich wie wir das bei der *Polymorphie* gesehen haben, über eine Liste von Objekten iteriert werden. Der Vorteil ist, dass dazu keine formale Schnittstelle vorhanden sein muss. Gleichzeitig ist dies aber auch ein Nachteil, so dass entsprechende Checks eingebaut werden sollten, um Fehler zu vermeiden.

Das *Duck-Typing* entspricht der Idee der oben erklärten Interfaces bei Sprachen wie Java oder C#.

Wie wird Duck-Typing verwendet?

Beim Duck-Typing werden beliebige Objekte in einer Liste gesammelt. Wird die Liste traversiert (mit einer for-Schleife), kann bei Objekten mit einer geforderten Methode - hier z.B. `swim` - diese ausgeführt werden.

Beispiel: Umsetzung einer pseudo Schnittstelle mit Duck-Typing

```
class People:
    def swim(self):
        print('schwimmen wie ein Mensch')

class Tree:
    def swim(self):
        print('schwimmen wie ein Baumstamm')

class Aeroplane:
    def swim(self):
        print('schwimmen wie ein Wasserflugzeug')

class Machine:
    pass # kann nicht schwimmen


if __name__ == '__main__':
    print('Deno Polymorphie mittels Interface (durch Duck-Typing)')

    container =  []
    container.append(People())
    container.append(Tree())
    container.append(Aeroplane())
    container.append(Machine())

    for obj in container:
        if hasattr(obj, 'swim'): # sicherstellen, dass nur Objekte mit einer
            obj.swim()           # Methode swim aufgerufen werden.
```

Das Programm liefert folgende Ausgabe:

```
schwimmen wie ein Mensch
schwimmen wie ein Baumstamm
schwimmen wie ein Wasserflugzeug
```

Der Grund dafür liegt darin, dass Python als dynamische Sprache bei Methoden mit derselben Signatur so tut, also ob dafür eine Schnittstelle vorhanden wäre. Das passiert automatisch und implizit (=versteckt). Wie bereits erwähnt, wird daher in diesem Zusammenhang von einer *informellen Schnittstelle* gesprochen:

Beispiel: Informelle Schnittstelle

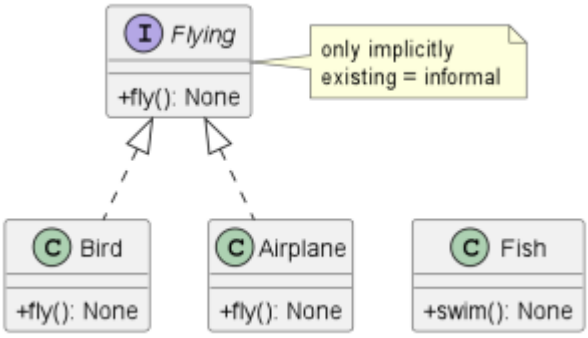

Abb: Die Schnittstelle `Flying` existiert nur implizit und ist daher informell

---

Quellen: Duck-Typing in Python