Informatik I: Einführung in die Programmierung

14. Objekt-orientierte Programmierung: Aggregierung, Properties, Invarianten, Datenkapselung

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

21.12.2021

1 Aggregierung

Zusammengesetzte Objekte

- Oft sind Objekte aus anderen Objekten zusammengesetzt.
- Methodenaufrufe auf ein zusammengesetztes Objekt führen meist zu Methodenaufrufen auf eingebetteten Objekten.
- Beispiel: ein zusammengesetztes 2D-Objekt, das andere 2D-Objekte enthält, z.B. einen Kreis und ein Rechteck.

Die Klasse Composite (1)

- Jede Instanz ist ein 2D-Objekt, aber eine Position macht keinen Sinn.
- Zusätzlich hat jede Instanz als Attribut eine Liste von 2D-Objekten.

newgeoclasses.py (1)

```
@dataclass
class Composite(TwoDObject):
    contents : list[TwoDObject] = field(init= False)

    def __post_init__(self):
        this.contents = []

    def add(self, *objs : list[TwoDObject]):
        self.contents.extend(objs)

    def rem(self, obj : TwoDObject):
        self.contents.remove(obj)
   ...
```

Die Klasse Composite (2)

- Die Methoden `size_change`, `move` und `position` werden überschrieben.
- Wir wälzen das Ändern und Verschieben des zusammengesetzten Objektes auf die Einzelobjekte ab: Delegieren.

`newgeoclasses.py (2)`

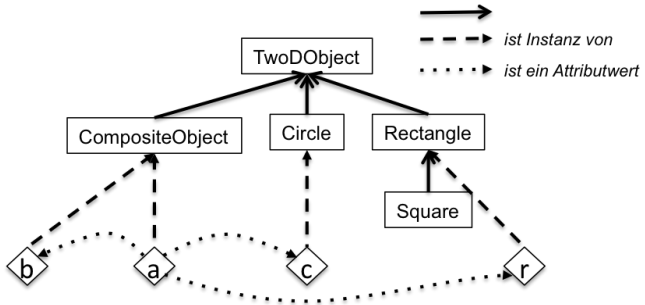
```
    def size_change(self, percent: float):
        for obj in self.contents:
            obj.size_change(percent)

    def move(self, xchange: float, ychange: float):
        for obj in self.contents:
            obj.move(xchange, ychange)

    def position(self):
        if self.contents:
            return self.contents[0].position()
        return super().position()
```

Die Klasse Composite (3)

Python-Interpreter

```
>>> c = Circle(x=1,y=2); r = Rectangle(height=10,width=10)
>>> a = Composite()
>>> a.add([r, c])
>>> a.size_change(200)
>>> r.area()
400.0
>>> a.move(40,40)
>>> a.position()
(40, 40)
>>> c.position()
(41, 42)
>>> b = Composite()
>>> a.add([b])
>>> a.move(-10, -10)
>>> b.position()
```

Vererbung und Komposition



2 Properties

Zugriff auf Attribute kontrollieren: Getter und Setter

- Ziel ist die Kontrolle über das Abfragen und Setzen von Attributwerten.
  - Invarianten zwischen Attributwerten sollen respektiert werden.
    Es soll nicht möglich sein, unsinnige Attributwerte zu setzen.
  - Der Zustand eines Objekts soll gekapselt werden.
- In anderen Sprachen können Attribute als privat deklariert werden.
  - Nur Methoden des zugehörigen Objekts können sie lesen bzw. ändern.
  - Sie sind unsichtbar für Objekte anderer Klassen.

  $\Rightarrow$ Datenkapselung; Invarianten können garantiert werden.
- Für den Zugriff durch andere Objekte werden (häufig) Getter- und (seltener) Setter-Methoden bereitgestellt.
  - Eine Getter-Methode liest ein privates Attribut.
  - Eine Setter-Methode schreibt ein privates Attribut.
- In Python sind Attribute im wesentlichen *öffentlich*, aber sie können durch Getter und Setter als Properties geschützt werden.

Datenkapselung und Invarianten

Definition: Dateninvariante

Eine Dateninvariante ist eine logische Aussage über die Attribute eines Objekts, die während der gesamten Lebensdauer des Objekts erfüllt sein muss.

- Der Konstruktor muss die Dateninvariante sicherstellen.
- Die Methoden müssen die Dateninvariante erhalten.
- Unbewachtes Ändern eines Attributs kann die Dateninvariante zerstören.

Definition: Datenkapselung

Attribute (Objektzustand) können nicht direkt gelesen oder geändert werden.

- Die Interaktion mit einem Objekt geschieht nur durch Methoden.
- Die Implementierung (Struktur des Objektzustands) kann verändert werden, ohne dass andere Teile des Programms geändert werden müssen.

Beispiel Invariante: Radius eines Kreises

- **Invariante**
  Das Attribut `radius` der Klasse `Circle` soll immer größer als Null sein.
- **Regel 1:** Jede Invariante **muss** im docstring der Klasse dokumentiert sein!

```
@dataclass
class Circle(TwoDObject):
    '''Represents a circle in the plane.

    Attributes:
        radius: a number indicating the radius of the circle
        x, y: inherited from TwoDObject

    Invariants:
        radius > 0
    '''
    radius : float
```

Beispiel Invariante: Radius eines Kreises

- Der docstring kann Verletzungen der Invariante nicht verhindern...
- **Regel 2**: Der Konstruktor muss die Einhaltung der Invariante prüfen!
- Die Prüfung geschieht durch eine Assertion in einer spezielle Methode `__post_init__`. Verletzung führt zu einer Exception (Ausnahme).
- `__post_init__` wird automatisch bei Konstruktion einer Instanz einer Datenklasse aufgerufen.

```
@dataclass
class Circle(TwoDObject):
    ...
    radius : float

    def __post_init__(self):
        assert self.radius > 0, "radius␣should␣be␣greater␣than␣0"
```

Was passiert?

- Bei falschem Aufruf des Konstruktors wird eine Exception ausgelöst.

Python-Interpreter

```
>>> c = Circle (x=10,y=20, radius=-3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../properties.py", line 46, in __init__
    assert radius > 0, "radius should be greater than 0"
AssertionError: radius should be greater than 0
```

Beispiel: Radius eines Kreises

- Ein böswilliger Mensch kann immer noch folgenden Code schreiben:

```
c = Circle(x=20, y=20, radius=5)
c.radius = -3  ## object invariant broken
```

- **Regel 3**: Das Attribut `radius` muss als Property ohne Setter definiert werden!

```
@dataclass
class Circle(TwoDObject):
    ...
    __radius : float ## Namen von Properties beginnen mit __

    def __post_init__(self):
        assert self.__radius > 0, "radius␣should␣be␣greater␣than␣0"

    @property
    def radius (self) -> float:
        return self.__radius
```

Was passiert?

- Der Attributwert für den Radius wird im Feld `__radius` des Objekts gespeichert.
  Felder, deren Name mit `__` beginnt, sind von außen nicht ohne weiteres zugreifbar!
- `radius` ist eine normale Methode, der Getter für `radius`.
- Die Dekoration mit `@property` bewirkt, dass `radius` wie ein Attribut verwendet werden kann.
- Ein Attributzugriff `c.radius` wird als Methodenaufruf `c.radius()` interpretiert.

Python-Interpreter

```
>>> c = Circle (x=10,y=20, radius=3)
>>> c.radius
3
>>> c.x = -3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Zusammenfassung Invariante

Eine (Daten-) Invariante ist eine logische Aussage über die Attribute eines Objekts, die während der gesamten Lebensdauer des Objekts erfüllt sein muss.

Regeln zu Dateninvarianten

1. Jede Invariante muss im docstring der Klasse dokumentiert sein!
2. Der Konstruktor muss die Einhaltung der Invariante prüfen!
3. Die Attribute, die in der Invariante erwähnt werden, müssen als Properties ohne Setter definiert werden!

Beispiel: Datenkapselung

Aufgabe

Ein Zeichenprogramm verwendet Punkte in der Ebene. Eine wichtige Operation auf Punkten ist die Drehung (um den Ursprung) um einen bestimmten Winkel.

Erster Versuch

```
@dataclass
class Point2D:
    x : float
    y : float

    def turn(self, phi : float):
        self.x, self.y = (self.x * cos(phi) - self.y * sin(phi)
                          ,self.x * sin(phi) + self.y * cos(phi))
```

Was passiert?

Python-Interpreter

```
>>> pp = Point2D(1,0)
>>> pp.x, pp.y
(1, 0)
>>> pp.turn(pi/2)
>>> pp.x, pp.y
(6.123233995736766e-17, 1.0)
>>> pp.y = -1
>>> pp.turn (pi/2)
>>> pp.x, pp.y
(1.0, 0.0)
```

Beobachtungen

- Das Interface von `Point2D` Objekten besteht aus den Attributen `x`, `y` und der Methode `turn()`.
- Jeder Aufruf von `turn()` erfordert vier trigonometrische Operationen (naja, mindestens zwei), die aufwändig zu berechnen sind.
- Möglichkeit zur Vermeidung der trigonometrischen Operationen:
  Ändere die Datenrepräsentation von rechtwinkligen Koordinaten (x, y) in Polarkoordinaten (r, ϑ). In Polarkoordinaten entspricht eine Drehung um φ der Addition der Winkel $\vartheta + \varphi$.
- Aber: das Interface soll erhalten bleiben!
- Ein Fall für Datenkapselung mit Gettern **und** Settern!
- (keine Invariante: `x` und `y` sind beliebige `float` Zahlen!)

Datenkapselung: Änderung der Repräsentation ohne Änderung des Interface


```
@dataclass
class PointPolar:
    x : InitVar[float]
    y : InitVar[float]

    def __post_init__ (self, x:float, y:float):
        self.__r = sqrt (x*x + y*y)
        self.__theta = atan2 (y, x)

    def turn (self, phi:float):
        self.__theta += phi
    ...
```

- `x` und `y` definieren nur die Parameter für den Konstruktor (Effekt von `InitVar`)
- Interne Repräsentation durch Polarkoordinaten
- Interne Attribute `__r` und `__theta` von außen nicht zugreifbar

Datenkapselung: Interface rechtwinklige Koordinaten


```
@property
def x (self) -> float:
    return self.__r * cos (self.__theta)
@property
def y (self) -> float:
    return self.__r * sin (self.__theta)
@x.setter
def x (self, x : float):
    self.__post_init__ (x, self.y)
@y.setter
def y (self, y : float):
    self.__post_init__ (self.x, y)
```

- Definition der Getter wie gehabt.
- Definition der Setter dekoriert mit `@x.setter`, wobei `x` der Propertyname ist.
- Methodendefinition für den Propertynamen mit einem Parameter (+ self).
- Eine Zuweisung `p.x = v` wird interpretiert als Methodenaufruf `p.x(v)`.

Was passiert? Exakt das Gleiche wie mit `Point2D`!

Python-Interpreter

```
>>> pp = PointPolar(1,0)
>>> pp.x, pp.y
(1, 0)
>>> pp.turn(pi/2)
>>> pp.x, pp.y
(6.123233995736766e-17, 1.0)
>>> pp.y = -1
>>> pp.turn (pi/2)
>>> pp.x, pp.y
(1.0, 0.0)
```

Weitere Möglichkeiten

- Intern könnte der Punkt **beide** Repräsentationen unterstützen.
- Nur die jeweils benötigte Repräsentation wird berechnet.
- Transformationen werden immer in der günstigsten Repräsentation ausgeführt:
  Rotation in Polarkoordinaten, Translation in rechtwinkligen Koordinaten, usw.

3 Operator-Überladung

- Arithmetische Operatoren
- Vergleichsoperatoren

Überladung von Operatoren

- Ein Operator ist überladen (operator overloading), wenn dieser Operator je nach Typ der Argumente (und ggf. dem Kontext) unterschiedlich definiert ist.
- Traditionell sind die arithmetischen Operatoren in vielen Programmiersprachen für alle numerischen Typen überladen.
- In Python sind außerdem die Operatoren „+“ und „*“ für Strings überladen.
- Für gewisse Operatoren können wir Überladung selbst definieren!
- Überladung ist immer mit Vorsicht zu geniessen:
  - Im Programmtext ist es nicht mehr offensichtlich, welcher Code ausgeführt wird, wenn überladene Operatoren vorkommen.
  - Eine Überladung darf nicht “die Intuition” eines Operators verletzen.
  - Beispiel: „+“ (auf Zahlen) hat Eigenschaften wie Kommutativität, Assoziativität, 0 als neutrales Element, etc, die durch Überladung nicht gestört werden sollten.

Beispiel: Addition für 2D-Punkte

point2d.py (1)

```
class Point2D:
    ...
    def __add__ (self, other):
        return Point2D (self.x + other.x, self.y + other.y)
```

- Die dunder[1] Methode `__add__` definiert die Überladung des „+“-Operators.
- Wenn `pp = Point2D (...)`, dann wird eine “Addition” `pp + v` als Methodenaufruf `pp.__add__(v)` interpretiert.
- Was fehlt hier?
- Was passiert, wenn `other` keine Instanz von `Point2D` ist?

[1] dunder = double underline

Beispiel: Addition für 2D-Punkte

point2d.py

```
class Point2D:
    ...
    def __add__ (self, other : Point2D):
        if isinstance (other, Point2D):
            return Point2D (self.x + other.x, self.y + other.y)
        else:
            raise TypeError ("Cannot␣add␣Point2D␣and␣" + str (type (other)))
```

- Der Funktionsaufruf `isinstance (other, Point2D)` testet, ob `other` eine Instanz von `Point2D` ist.
- Falls nicht, wird hier eine Exception erzeugt.

Beispiel: Multiplikation für 2D-Punkte

mit den dunder Methoden `__mul__` und `__rmul__`

UNI FREIBURG

point2d.py

```
class Point2D:
    ...
    def __mul__ (self, other : Union[Point2D,numbers.Number]):
        if isinstance (other, Point2D):     # scalar product
            return self.x * other.x + self.y * other.y
        elif isinstance (other, numbers.Number): # scalar multiplication
            return Point2D (other * self.x, other * self.y)
        else:
            raise TypeError ("Cannot␣multiply␣Point2D␣and␣" + str (type (other)))

    def __rmul__ (self, other : numbers.Number):
        if isinstance (other, numbers.Number):
            return Point2D (other * self.x, other * self.y)
        else:
            raise TypeError ("Cannot␣multiply␣" + str (type (other)) + "␣and␣Point2D")
```

Was passiert?

Python-Interpreter

```
>>> p1 = Point2D (1,0)
>>> p1.x, p1.y
(1, 0)
>>> p2 = p1 * 42 # multiply p1 with a number
>>> p2.x, p2.y # yields a point
(42, 0)
```

- `p1 * 42` entspricht `p1.__mul__(42)`; `other` ist eine Zahl

Multiplikation (2)

Python-Interpreter

```
>>> w = p1 * p2 # multiply two points
>>> w # yields a number
42
```

- `p1 * p2` entspricht `p1.__mul__(p2)`; `other` ist eine Instanz von `Point2D`

Multiplikation (3)

Python-Interpreter

```
>>> p3 = 3 * p1 # multiply a number with a point
>>> p3.x, p3.y # yields a point
(3, 0)
```

- `3 * p1` entspricht ...
- `3.__mul__(p1)` ... — *im Prinzip; kann so nicht eingegeben werden*
- aber der Typ `int` kann nicht mit einem `Point2D` multiplizieren. Daher liefert dieser Versuch den Wert `NotImplemented`.
- Daraufhin versucht es Python mit vertauschten Operanden ...
- `p1.__rmul__(3)` ... was ein Ergebnis liefert.
- Die arithmetischen Operatoren +, *, -, / und % können nach dem gleichen Muster überladen werden.

Vergleichsoperatoren

- Die Vergleichsoperatoren == und `!=` können mit den dunder Methoden `__eq__` und `__ne__` definiert werden.
- Sinnvolle Anwendung von Überladung, da für jeden Typ eine andere Implementierung der Gleichheit erforderlich ist!

Vergleich: `__eq__`, `__ne__`

- `obj.__eq__(other)`
  - Auswertung von `obj == other`.
  - Auswertung von `other == obj`, falls `other` keine `__eq__` Methode besitzt.
- `obj.__ne__(other)`
  Auswertung von `obj != other` (oder `other != obj`).
- Der Aufruf von `!=` gibt automatisch das Gegenteil vom Aufruf von == zurück, außer wenn == das Ergebnis `NotImplemented` liefert. Es reicht also, `obj.__eq__(other)` zu implementieren.

- Ohne diese Methoden werden Objekte nur auf Identität verglichen, d.h. `x == y` gdw. `x is y`.

Gleichheit für 2D Punkte

Equality

```
@dataclass
class Point2D:
    ...
    def __eq__ (self, other):
        return (isinstance(other, Point2D) and
                self.x == other.x and self.y == other.y)
```

- Datenklassen haben automatisch eine Methode `__eq__`, falls nicht explizit eine definiert wird.
- Das Beispiel zeigt die Methode `__eq__`, wie sie für die Datenklasse `Point2D` automatisch erzeugt wird.

Vergleich: `__ge__`, `__gt__`, `__le__`,`__lt__`

- `obj.__ge__(other)`
  - Zur Auswertung von `obj >= other`.
  - Zur Auswertung von `other <= obj`, falls `other` über keine `__le__`-Methode verfügt.
- `obj.__gt__(other)`, `obj.__le__(other)`, `obj.__lt__(other)`:
  Analog für `obj > other` bzw. `obj <= other` bzw. `obj < other`.
- Auch die Vergleichsmethoden können automatisch durch die Datenklasse erzeugt werden, wenn `order=True` angegeben wird:

```
@dataclass(order=True)
class Point2D:
    x : float
    y : float
```

4 Zusammenfassung

Zusammenfassung

- Aggregierung liegt vor, falls Attribute von Objekten selbst wieder Objekte sind.
- Properties erlauben die Realisierung von Invarianten und Datenkapselung. Attributzugriffe werden über Getter und Setter (Methoden) abgewickelt.
- Überladung liegt vor, wenn ein Operator die anzuwendende Operation anhand des Typs der Operanden bestimmt.
- Python verwendet dunder Methoden zur Implementierung der Überladung von Operatoren.