Informatik I: Einführung in die Programmierung

13. Objekt-orientierte Programmierung: Aggregierung, Properties, Invarianten, Datenkapselung, Operatoren

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

10. Dezember 2025

1 Aggregierung

Zusammengesetzte Objekte

- Oft sind Objekte aus anderen Objekten zusammengesetzt.
- Methodenaufrufe auf ein zusammengesetztes Objekt führen meist zu Methodenaufrufen auf den 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.
- Zusätzlich hat jede Instanz als Attribut eine Liste von 2D-Objekten.
- Sie wird durch `default_factory= list` mit der leeren Liste initialisiert.

newgeoclasses.py (1)

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

    def add(self, obj : TwoDObject):
        self.contents.append(obj)

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

Die Klasse Composite (2)

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

newgeoclasses.py (2)

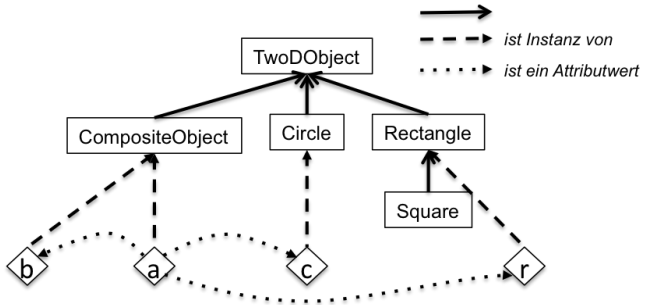
```
    def size_change(self, percent: float):
        for obj in self.contents:
            obj.size_change(percent)

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

Die Klasse Composite (3)

Python-Interpreter

```
>>> c = Circle(x=1,y=2); r = Rectangle(height=10,width=10)
>>> a = Composite()
>>> a.add(r)
>>> a.add(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 Getter- und 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 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

- **Regel 2**: Das Attribut muss mit Getter und Setter definiert werden.

```python
@dataclass
class Circle(TwoDObject):
    __radius : InitVar[float]

    def __post_init__(self, radius : float):
        self.set_radius(radius)

    def get_radius(self) -> float:
        return self.__radius

    def set_radius(self, radius: float):
        assert radius > 0, "Radius should be greater than zero"
        self.__radius = radius
```

Beispiel Invariante: Radius eines Kreises


```
    __radius : InitVar[float]
```

Definiert das (versteckte) Feld, in dem der Wert gespeichert wird.
Felder, deren Name mit __ beginnt, sind von außen nicht ohne weiteres zugreifbar!

```
    def __post_init__(self, radius : float):
        self.set_radius(radius)
```

Attribute vom Typ `InitVar` werden an die `__post_init__` Methode übergeben.

```
    def get_radius(self) -> float:
        return self.__radius
```

Definiert den Getter für die Property `radius`, eine normale Methode.

Beispiel Invariante: Radius eines Kreises


```
    def set_radius(self, radius: float):
        assert radius > 0, "Radius should be greater than zero"
        self.__radius = radius
```

Definiert den Setter der Property `radius`.

- **Regel 3**: Der Setter muss die Einhaltung der Invariante prüfen!
- Die Prüfung geschieht durch eine Assertion im Setter. Verletzung führt zu einer Exception (Ausnahme).

Was passiert?

- Bei falschem Aufruf des Konstruktors wird eine Exception ausgelöst.

```
>>> c = Circle (x=10,y=20, radius=-3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Circle' is not defined
```

Beispiel: Radius eines Kreises

- Ein böswilliger Mensch kann immer noch folgenden Code schreiben:

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

Stattdessen schreiben wir

```
c.set_radius(-3)
```

Invariante für unveränderliche Objekte


```python
@dataclass(frozen=True)
class Vehicle:
    wheels : int

    def __post_init__(self):
        assert self.wheels > 0, "nr of wheels should be greater than zero"
```

- **Regel 4**: Bei unveränderlichen Objekten muss der Konstruktor die Einhaltung der Invariante prüfen!
- Die Prüfung geschieht durch eine Assertion in einer speziellen Methode `__post_init__`. Verletzung führt zu einer Exception (Ausnahme).
- `__post_init__` wird automatisch nach Setzung der Attribute bei der Konstruktion einer Instanz einer Datenklasse aufgerufen.

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. Bei veränderlichen Objekten muss (nur) der Setter die Einhaltung der Invariante prüfen!
3. Bei unveränderlichen Objekten muss der Konstruktor die Einhaltung der Invariante prüfen!

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 ohne Weiteres zugreifbar

Datenkapselung: Interface rechtwinklige Koordinaten


```
    def get_x (self) -> float:
        return self.__r * cos (self.__theta)
    def get_y (self) -> float:
        return self.__r * sin (self.__theta)
    def set_x (self, x : float):
        self.__post_init__ (x, self.get_y())
    def set_y (self, y : float):
        self.__post_init__ (self.get_x(), y)
```

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

Python-Interpreter

```
>>> pp = PointPolar(1,0)
>>> pp.get_x(), pp.get_y()
(1, 0)
>>> pp.turn(pi/2)
>>> pp.get_x(), pp.get_y()
(6.123233995736766e-17, 1.0)
>>> pp.set_y(-1)
>>> pp.turn (pi/2)
>>> pp.get_x(), pp.get_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 Operatoren

- 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.
- Anwendung: arithmetische Operatoren auf numerischen Typen, die für jeden Typ anders implementiert sind.
- In Java oder C++ (z.B.) geschieht die Auflösung der Überladung vor der Laufzeit (z.B. durch einen Compiler) aufgrund der Argumenttypen.
- 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.

Operatoren in Python

- In Python ist die Lage anders, weil alles erst zur Laufzeit passiert!
- Es hat den Anschein, dass die Operatoren „+“ und „*“ überladen sind, weil sie für verschiedene Argumenttypen (`int`, `str`, ...) definiert sind.
- In Wirklichkeit handelt es sich um eine Anwendung von dynamic dispatch, weil die Auswahl allein durch den Laufzeittyp des ersten Operanden geschieht.
- D.h. intern werden „+“ und „*“ als Methodenaufrufe behandelt!
- Für gewisse Operatoren können wir diese Methoden überschreiben und sie dadurch selbst definieren!

Beispiel: Addition für 2D-Punkte

point2d.py (1)

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

- Die dunder[1] Methode `__add__` definiert den „+“-Operator.
- Generell wird eine “Addition” `pp + v` als Methodenaufruf `pp.__add__(v)` interpretiert (für `pp = Point2D (...)`).
- Was fehlt im Code?
- 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: Any) -> 'Point2D':
        match other:
            case Point2D(x, y):
                return Point2D (self.x + x, self.y + y)
            case _:
                raise TypeError ("Cannot add Point2D and " + str (type (other)))
```

- Teste mit Patternmatching, 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__`

point2d.py

```
class Point2D:
    # ...
    def __mul__ (self, other: Any) -> 'Point2D | float':
        match other:
            case Point2D(x, y):
                return self.x * x + self.y * y
            case int() | float():
                return Point2D (other * self.x, other * self.y)
            case _:
                raise TypeError ("Cannot multiply Point2D and " + str (type (other)))
```

- Bei der Multiplikation gibt es mehr Freiheitsgrade: wir können Punkte multiplizieren, Punkte mit Zahlen oder auch Zahlen mit Punkten…

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)` — *Leerzeichen beachten*
- 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 definiert werden.

Multiplikation von rechts

point2d.py

```
class Point2D:
    # ...
    def __rmul__ (self, other: Any) -> 'Point2D':
        match other:
            case int() | float():
                return Point2D (other * self.x, other * self.y)
            case _:
                raise TypeError ("Cannot multiply " + str (type (other)) + " and Point2D")
```

Vergleichsoperatoren

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

Vergleich von Objekten: `__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 (type(other) is 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.
- Attributzugriffe über Getter und Setter erlauben die Realisierung von Invarianten und Datenkapselung.
- Überladung liegt vor, wenn ein Operator die anzuwendende Operation anhand des Typs der Operanden bestimmt.
- Python verwendet dunder Methoden zur Implementierung von Operatoren und simuliert Überladung durch dynamic dispatch.