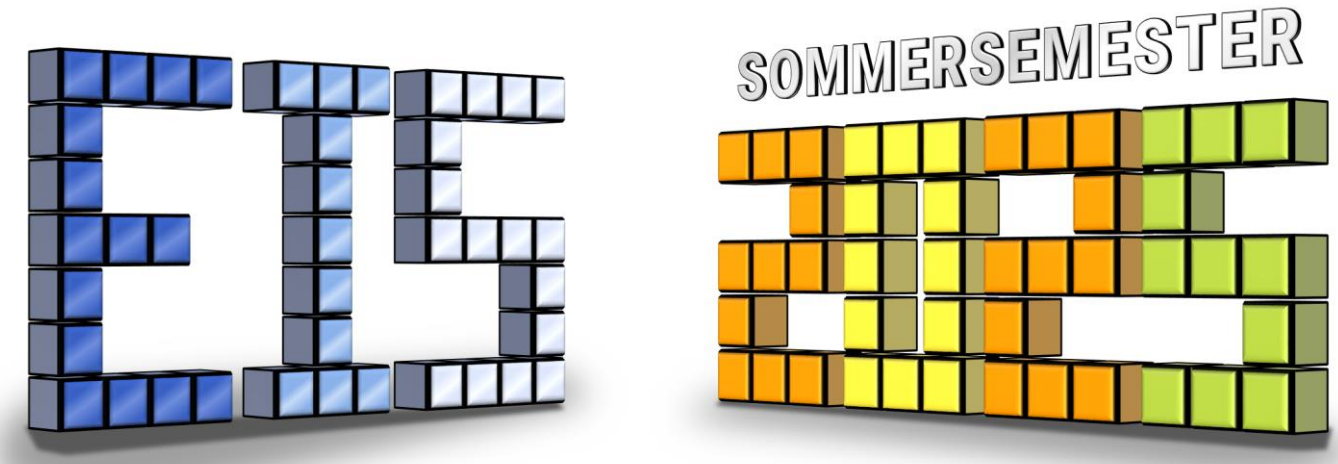
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #7

Objekt-Orientierte Programmierung

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de

Techniken

Strukturierung von Programmen

- Prozedural
- Objekt-orientiert
- Funktional
- Meta-Programmierung

Funktionale Programmierung

Idee

Programme aus Teilen zusammensetzen

- Code-Schnipsel als „Variablen“
- Dynamisch (d.h., zur Laufzeit) zusammensetzen
- Mehr Flexibilität

Bislang: Unterprogramme („Funktionen“)

- Werden statisch zu einem Programm zusammengesetzt (Unterprogrammaufruf)
- Jetzt neu: Zusammensetzung kann zur Laufzeit bestimmt werden

Funktionale Programmierung

Abgrenzung: „Pure Functional Programming“

- Spezielle Variante von FP
  - Richtung *innerhalb* von FP
    - manchmal synonym verwendet
  - Mathematisch angehaucht
- Prinzipien
  - Keine (oder eingeschränkte) Änderung von Variablen
    - „Avoiding mutable state“
  - Impliziert: Rekursion statt Schleifen

Später mehr dazu

Funktionale Abstraktionen

Unterstützung durch Programmiersprache

- Wir brauchen eine „Abstraktion“
  - Sprachelement / Feature
  - Einfach zu benutzen und (halbwegs) sicher

Zwei Möglichkeiten

- Dynamische Übersetzung/Evaluation **(allgemeiner)**
  - Code wird zur Laufzeit neu Erzeugt
  - Möglich in z.B. Python, LISP, C#
- Funktionszeiger (statisch, **eingeschränkter**)
  - Verweise auf bestehende Unterprogramme als Variablen
  - Möglich in allen modernen Sprachen (z.B. C/C++, Scala)

Code zur Laufzeit erzeugen (Python)

```
# Variante 1a: Dynamische Auswertung (via Strings)
x = 21
# Output: 42
print( eval('x * 2') )

# Variant 1b: Dynamische Ausführung (via Strings)
# Output: Hello World
exec('print("Hello World")')

# Variant 1c: Code objects durch Kompilieren von Strings
code_obj = compile('print("Hello World")', '<string>', 'exec')
dis.dis(code_obj) # "import diss" needed
# Output:
#   0           0 RESUME                   0
#   1           2 PUSH_NULL
#               4 LOAD_NAME                0 (print)
#               6 LOAD_CONST               0 ('Hello World')
#               8 PRECALL                  1
#              12 CALL                     1
#              22 POP_TOP
#              24 LOAD_CONST               1 (None)
#              26 RETURN_VALUE
```

Funktionszeiger (Python)

```python
# Funktion definieren
def add42(x: int) -> int:
    return x + 42

# Typdefinition für Funktionsvariable
IntFunc: TypeAlias = Callable[[int], int]

# Funktionsvariable
add_a_suitable_amount: IntFunc = add42

# Benutzung
print(add_a_suitable_amount(1337))

# “Interessantere” Anwendung
def apply_to_list(some_list: List[int] operation: IntFunc) -> None:
    for i in range(len(some_list)):
        some_list[i] = operation(some_list[i])

apply_to_list([1, 2, 3], add42) # -> [43, 44, 45]
```

Funktionszeiger in C++

Funktionszeiger / Funktionsobjekte (C/C++)

```
// Funktion definieren
int add42(int x) {
    return x + 42;
}

typedef int(*IntToIntFunc)(int);

int main() {
   // Funktionszeiger
   IntToIntFunc add_a_suitable_amount = &add42;

   // Benutzung: Dereferenzierung
   cout << (*add_a_suitable_amount)(1337);

   return 0;
}
```

Funktionszeiger

Funktionszeiger / Funktionsobjekte (Pascal)

```
(* Funktion definieren *)
FUNCTION add42(x : INTEGER) : INTEGER
BEGIN
    return x + 42;
END

(* Object Pascal / Delphi *)
TYPE IntToIntFunc = FUNCTION(x : INTEGER) : INTEGER;

(* Variablendeklaration *)
VAR add_a_suitable_amount : IntToIntFunc;

BEGIN
   (* Funktionszeiger *)
   add_a_suitable_amount = add42;
   (* Benutzung *)
   WriteLn(add_a_suitable_amount(1337));
END
```

Funktionsreferenzen in Scala

```scala
// Funktion definieren
def add42 = (x : Int) => {x + 42;}

// Typdefinition für Funktionstyp
type IntToIntFunc = (x : Int) => Int;

@main def mainProgramm = {
    // Funktionszeiger
    var add_a_suitable_amount: IntToIntFunc = add42;
    // Benutzung
    println(add_a_suitable_amount(1337));
}
```

Objektorientierte Programmierung

Drei Ideen kombiniert…

Objektorientierte Programmierung

- **Abstrakte Datentypen**
  - Daten der „Objekte“ gekapselt mit Unterprogrammen („Methoden“)
  - Schnittstellen definieren („Klassen“)
- **Polymorphie:** „Dynamic Dispatch“
  - Automatisch Auswahl des geeigneten Codes
  - Auswahlkriterium: Typ des Objektes (Klasse)
    - „Dynamic Dispatch on Object Type“
- **Vererbung:** mehr Wiederverwendung
  - Hierarchische Faktorisierung von Code & Datenstrukturen
  - „Subtyping“ für statische Typprüfung

Drei Ideen kombiniert…

Objektorientierte Programmierung

- Abstrakte Datentypen
  - Prozedurales Muster
- Polymorphie: „Dynamic Dispatch“
  - Funktionales Muster
  - DRY für Benutzer/innen der Objekte
- Vererbung für Wiederverwendung / Strukturierung
  - „Kompression“ des Codes
  - DRY für Objekt selbst

Herausforderungen

Objektorientierte Programmierung

- Abstrakte Datentypen
  - → Gute Schnittstellen
- Polymorphie: „Dynamic Dispatch“
  - → Schnittstellen müssen generisch über verschiedene Datentypen sein
- Vererbung für Wiederverwendung / Strukturierung
  - → Sinnvolle hierarchische Struktur in Schnittstellen sowie Code und Daten (u.U. herausfordernd)

OOP Praktisch

Disclaimer – Hinweis zu Übungen

Code-Beispiele in Folien

- Entwurfsideen für Zeichenprogramm als Beispiele
- Nicht unbedingt „beste Lösung“
  - Teilweise mehrere Alternativen betrachtet
  - Teilweise „konstruiert“ um Effekt zu zeigen
  - Eher Zoo von Ideen
- Für die Übungsaufgaben
  - Sie sollten (ruhig) Ihre eigene Lösung entwickeln
  - Kreativität / Erfahrung
  - Design der Vorlesung nicht unbedingt „optimal“ für das, was in den Übungen gefordert ist

ADTs in OOP

Abstrakte Datentypen in OO-Systemen

- Datentypen = „**Klassen**“
  - Instanzen = „Objekte der Klasse“
- Unterprogramme für Zugriff = „**Methoden**“
  - Objektmethoden: Zugriff auf ein Objekt
  - Klassenmethoden: Allgemeine UP ohne Objektbezug
- (Typische) Zugriffssteuerung
  - „**private**“: Zugriff nur innerhalb Codes der eigenen Klasse
  - „**protected**“: Zugriff auch für Nachfahren bei Vererbung
    - (Zusätzliche) Schnittstelle für Vererbung
  - „**public**“: Allgemeiner Zugriff
    - Öffentliche Schnittstelle

Abstrakte Datentypen (Python)

```
# Klasse definieren
class Rectangle:
    # Konstruktor
    def __init__(self, pos: Vector2d, width: float, height: float):
        self.__pos: Vector2d = pos     # privates Attribut
        self.__width: float = width    # privates Attribut
        self.__height: float = height # privates Attribut

    # öffentliche Methode
    def get_width(self) -> float:
        return self.__width

    # öffentliche Methode
    def draw(self, painter: QPainter) -> None:
        painter.drawRect(self.__pos.x, self.__pos.y,
                         self.__width, self.__height)
```

Abstrakte Datentypen (Python)

```python
# Klasse definieren
class Rectangle:
    # Konstruktor
    def __init__(self, pos: Vector2d, width: float, height: float):
        self.__pos: Vector2d = pos
        self.__width: float = width
        self.__height: float = height
        if not self.__check_consistent():
            raise RuntimeError("width and height must be positive")

    # private Methode: z.B. Konsistenzprüfung
    def __check_consistent(self) -> bool:
        return self.width >= 0 and self.height >= 0

    # halb-private ("protected") Methode
    def _get_lower_right_corner(r: Rectangle) -> Vector2d:
        return Vector2d(r.pos.x + r.width, r.pos.y + r.height)
```

ADTs in Python

Zugriffssteuerung

- Alles ist „**public**“
- Konvention
  - Bezeichner, die mit „`_`“ anfangen sollen „**protected**“ sein
  - Bezeichner, die mit „`__`“ anfangen sollen „**private**“ sein
- Abgrenzung
  - Bezeichner, die in „`__<name>__`“ eingeschlossen sind, sind „magic“-Methoden für Python-interne Zwecke
    - z.B. „`__init__`“ oder „`__add__`“

ADTs in Python

Objektbezogene und andere Methoden

- Objektbezogene Unterprogramme
  - Bekommen expliziten Paramter „`self`“ um auf Instanz zuzugreifen
- Ohne `self` kein Zugriff auf Objekte
  - `self` hat den dynamischen Typ der Klasse des Objektes
  - `MyPy`: Es wird i.d.R. kein Typ für `self` angegeben

ADTs in Python

Objektbezogene und andere Methoden

- Methoden ohne „`self`“?

```
class Example:
    @staticmethod
    def do_something() -> None:
        print("just a stand-alone function")

    @classmethod
    def do_something(cls) -> None:
        print("I belong to: " + cls.__name__)
```

  - **@staticmethod**: Unterprogramme in Klassen sortiert
    - Ohne „@staticmethod“ Aufruf nur mit Objekt möglich
  - **@classmethod**: Klasse als Parameter `cls` bei Aufruf
    - Nützlich sein, wenn Informationen aus Klasse nötig (r/w)

Dynamic Dispatch

Automatische Auswahl von Unterprogrammen

- Objektorientiertes Programmieren:
  - „Nachricht an Objekt sende“
  - „Tu etwas für mich – danke!“
  - (Synchron: Normalerweise nicht parallel/nebenläufig)
- Methodenaufruf (Python)
  `objekt.methode(parameter)`
  - Auswahl der richtigen Methode nach Typ von „`objekt`“

Methodenauswahl (Python)

```python
class Rectangle:
    # Felder pos (2f), width (f), height (f) – Details ausgelassen
    def draw(self, painter: QPainter):
        painter.drawRect(self.pos.x, self.pos.y, self.width, self.height)

class Circle:
    # Felder center (2f), radius (f) – Details ausgelassen
    def draw(self, painter: QPainter):
        painter.drawCircle(self.center.x, self.center.y, self.radius)

some_shape = Rectangle(…)
some_shape.draw(painter) # ruft Retangle.draw() auf
Rectangle.draw(some_shape, painter) # äquivalenter Syntax, gleicher Effekt
print(type(some_shape)) # -> Rectangle

some_shape = Circle(…)
some_shape.draw(painter) # ruft Circle.draw() auf
print(type(some_shape)) # -> Circle
```

Na wunderbar!

Python erlaubt polymorphen Methodenaufruf

- Das „richtige“ Unterprogramm wird automatisch ausgewählt
- „Richtig“ definiert als Methode der Klasse des Objektes

Wir können unser Projekt deutlich vereinfachen…

Zeichnen: Original Prozedural

```
def draw_circle(c: Circle, painter: QPainter) -> None:
    painter.drawCircle(c.center.x, c.center.y, c.radius)

def draw_rectangle(r: Rectangle, painter: QPainter) -> None:
    painter.drawRect(r.pos.x, r.pos.y, r.width, r.height)

def draw_star(r: Star, painter: QPainter) -> None:
    # …ommitted… (zu lang)


def render_scene(s: Scene, painter: QPainter) -> None:
    for shape in s.all_shapes:
        if instanceof(shape, Circle):
            draw_circle(shape, painter)
        elif instanceof(shape, Rectangle):
            draw_rectangle(shape, painter)
        elif instanceof(shape, Star):
            draw_star(shape, painter)
        else:
            assert False # I do not like this!
```

Zeichnen: Basic OOP

```
class Circle:
  def draw(self, painter: QPainter) -> None:
    painter.drawCircle(self.center.x, self.center.y, self.radius)

class Rectangle:
  def draw(self, painter: QPainter) -> None:
    painter.drawRect(self.pos.x, self.pos.y, self.width, self.height)

class Star:
   def draw(self, painter: QPainter) -> None:
     # …ommitted… (zu lang)


class Scene:
    def render(self, painter: QPainter) -> None:
        for shape in self.all_shapes:
            shape.draw(painter)
```

→ Einfacher, leichter erweiterbarer, weniger fehleranfällig

Was fehlt noch?

Potential für Verbesserungen

- „Ducktyping“ – keine statische Typprüfung
  - Das ist derzeit gerade der Witz bei der Sache :-)
- Keine Gemeinsamkeiten zwischen Typen
  - Jedes Primitiv muss „von Grund auf neu“ programmiert werden

Warum?

- Typprüfung: Schneller & sicherer
- Gemeinsamkeiten nur einmal Programmieren
  - Komplexe Frameworks (Historisches Beispiel OLE/COM)

Subtyping durch Vererbung

```
class Shape:
  def draw(self, painter: QPainter) -> None:
    pass # Tut nix!


class Circle(Shape): # „Erbt“ von Shape
  def draw(self, painter: QPainter) -> None:
    painter.drawCircle(self.center.x, self.center.y, self.radius)

class Rectangle(Shape): # „Erbt“ auch von Shape
  def draw(self, painter: QPainter) -> None:
    painter.drawRect(self.pos.x, self.pos.y, self.width, self.height)

class Star(Shape): # „Erbt“ auch von Shape
   def draw(self, painter: QPainter) -> None:
     # …ommitted… (zu lang)
```

→ Gemeinsamer „Obertyp“ (Oberklasse) `Shape`

Subtyping erlaubt Typprüfung

```
some_shape: Shape = … # Variable mit Oberklasse definiert

some_shape = Rectangle(…) # Zuweisung von Untertypen erlaubt!
some_shape.draw(painter)  # Es ist garantiert, dass draw existiert

some_shape = Circle(…)    # Zuweisung von Untertypen erlaubt!
some_shape.draw(painter) # Es ist garantiert, dass draw existiert

some_shape = int(…)       # MyPy meldet hier einen Fehler!
some_shape.draw(painter) # CPython würde einen Laufzeitfehler produzieren
```

Vererbung von Methoden

```
class Shape:
  def __init__(self, pos: Vector2d, width: float, height: float):
    self.__pos: Vector2d = pos
    self.__width: float = width
    self.__height: float = height

  def draw(self, painter: QPainter) -> None:
    pass # Tut nix!
```

Shape

```
class Circle(Shape): # „Erbt“ von Shape
  def __init__(self, pos: Vector2d, radius: float):
    super().__init__(self, pos, radius, radius)

  def draw(self, painter: QPainter) -> None:
    painter.drawEllipse(self.__pos.x, self.__pos.y, self.__width, self.__height)
```

Circle/Ellipse

```
class Rectangle(Shape): # „Erbt“ von Shape
    def __init__(self, pos: Vector2d, width: float, height: float):
        super().__init__(self, pos, width, height)

  def draw(self, painter: QPainter) -> None:
    painter.drawRect(self.__pos.x, self.__pos.y, self.__width, self.__height)
```

Rectangle

Gemeinsame „Infrastruktur“

```
class Shape:
  def __init__(self, pos: Vector2d, width: float, height: float):
    …                                                    # gemeinsam genutzt

  def get_width(self) -> float:                          # gemeinsam genutzt
    return self.__width

  def set_width(self, width: float) -> None:             # gemeinsam genutzt
    self.width = __width

  # …

  def move(self, x: float, y: float) -> None:            # gemeinsam genutzt
    self.__pos.x += x
    self.__pos.y += y

  def scale(self, fact: float) -> None:                  # gemeinsam genutzt
    self.__pos.x -= fact * self.__width / 2.0
    self.__pos.y -= fact * self.__height / 2.0
    self.__width *= fact
    self.__height *= fact

  def draw(self, painter: QPainter) -> None:                 # Leere Vorlage
    pass
```

Vererbung

Regeln für Vererbung in Python

- Vererbung definiert Typhierarchie
  - Direkte und indirekte Unterklassen sind Untertypen
  - Zuweisungskompatibel
  - Erben die ganze Schnittstelle
- Vererbung überträgt Methoden
  - Oberklassenmethoden werden in Unterklasse übernommen
  - Werden diese neu definiert („override“), dann gelten die spezielleren Methoden
- Speziell in Python: Struktur von Klassen
  - Im Konstruktor definiert (geerbter: Aufruf z.B. via `super()`)
  - Bei `@dataclass`es werden Felder mitvererbt

Griechische Philosphen

Definitionshierarchie

- „Genus proximum et differentia specifica“
  - Geht auf Aristoteles zurück
- Definition als Hierarchie (Baum oder DAG)
- Schrittweise Spezialisierung von Oberklassen
  - Offensichtlich uneindeutig, wenn keine Baumstruktur vorliegt
  - Trotzdem oft nützlich
  - Motivation für Vererbung
- Kein-Baum-Problem motiviert andere Ideen
  - Mehrfachvererbung, Interfaces, MixIns, Traits, etc.

Subtyping vs. Sum-Types

Subtypen

- Implementation von Summentypen im Sinne von „Man kann hier verschiedene Typen finden“
- Unterschied / Vorteil
  - Gemeinsamkeiten definiert
  - Geerbte Oberklasse = gemeinsame Schnittstelle
  - Datenfelder die immer Verfügbar sind
  - Methoden (vorher nicht modelliert), die immer verfügbar sind

Vorteile Vererbung

Aspekt: Subtyping

- Erlaubt strenge Typprüfung
- Sicherer (und in anderen Sprachen: viel schneller)

Aspekt: Methoden/Datenstruktur erben

- Weniger Code (hierarchische Komprimierung)
- Potentiell konsistenter, weniger fehleranfällig
  - Bei komplexen Schnittstellen sehr wertvoll, siehe „QWidget“
  - Jedes neue Widget komplett neu implementieren wäre extrem aufwendig
    - Siehe Übungen!

Nachteile Vererbung

Schlecht entworfene Vererbung problematisch

- Starke Koppelung der Komponenten
  - `QWidget` muss innerhalb von Qt benutzt werden
    - Sonst macht es keinen Sinn
  - „Das ganze Framework hängt mit drin“
    - Hier kein Problem, an anderer Stelle vielleicht schon
    - Man denke an kleine Routine, um Text zu formatieren
- Gefahr unübersichtlich und unflexibel zu werden

Goldene Entwurfsregel

- Vererbung = „Ist-ein“-Beziehung
  - Ausführlichere Diskussion folgt

Abstrakte Klassen

```
from abc import ABC, abstractMethod;

class Shape(ABC): #  ABC = „Abstract Base Class“ (in Python)
    @abstractmethod   #  darf man nicht aufrufen
    def draw(self, painter: QPainter) -> None:
        pass

class Circle(Shape): # „Erbt“ von Shape
  def draw(self, painter: QPainter) -> None:
    painter.drawCircle(self.center.x, self.center.y, self.radius)

class Rectangle(Shape): # „Erbt“ auch von Shape
  def draw(self, painter: QPainter) -> None:
    painter.drawRect(self.pos.x, self.pos.y, self.width, self.height)
```

Abstrakte Oberklassen (mit abstrakten Methoden)

- Stellt sicher, dass „Shape“ nicht instantiiert wird
  - Reine Vorlage für Nachfahren

Andere Programmiersprachen

Andere Programmiersprachen

Scala

Abstrakte Datentypen in Scala

```
// Klasse in Scala: Konstruktor in Definition
class Rectangle(var pos: Vector2d,
                var width: Float, var height: Float) {

  // Weiterer ("Auxilliary") Konstruktor (public ist "default" - also voreingstellt)
  def this(var ul: Vector2d, var lr: Vector2d) = {
    // Standard Konstruktor
    this(ul, lr.x - ul.x, lr.y - ul.y)
    // Weitere Aktionen möglich
  }

  def set_width(w: Float): Unit = {  // Änderung der Breite (auch public)
    width = w
  }

  // Konvention: Keine runden Klammer, falls Methode nichts ändert und nur Infos zurückliefert
  protected def get_lower_right_corner: Vector2d = {
    new Vector2d(pos.x + width, pos.y + height)
  }

  // Drei Zugriffslevel: private, protected und <gar nichts> = public (kein Schlüsselwort)

}
```

Klassenmethoden?

```
// normale Instanz (zur Abgrenzung)
val unit_rect: Rectangle(Vector2d(0, 0), 1, 1);

// Scala "Compagnion-Objects" (fixe Instanzen mit eigenen Methoden, Feldern, etc.)
object Rectangle {
  // Klassenvariable
  var uniform_rect_counter: Int = 0;

  // Klassenmethode
  def make_uniform_rect(size: Float) = {
    uniform_rect_counter += 1; // I will make one more of those! (just an example)
    val hsize: Float = size/2.0f; // f suffix = Float (statt Double)
    new Rectangle(Vector2d(-hsize, -hsize), size, size);
  }

}
```

(Einfache) Vererbung in Scala

```
// Abstrakte Oberklasse („abstract“ optional: verhindert Instantiierung)
abstract class Shape() {
  def draw(g: Graphics): Unit = {…}
}

class Rectangle(var pos: Vector2d,
                var width: Float, var height: Float) extends Shape {

  override def draw(g: Graphics): Unit = {…} // Override überschreibt geerbte M.

}

class Circle(var center: Vector2d,
             var radius: Float) extends Shape {

  override def draw(g: Graphics): Unit = {…} // Override übers. geerbte Methode

}
```

OOP in Scala vs. Python

Strenge Typprüfung

- Nur Subtypen erlaubt
  - In Python: Code läuft, wenn alle Methoden zur Laufzeit da sind
  - In Scala: Code kompiliert, wenn Typen Methoden garantieren
- Vererbung damit wichtiger
  - Daher gibt es noch weitere Mechanismen
  - Später im Kapitel: Advanced OOP

OOP in Scala vs. Python

Vererbung von Code und Daten

- Wenn man „klassische“ Vererbung nutzt
  - Alle Methoden werden von Basisklasse übernommen
  - Alle Datenfelder werden von Basisklasse übernommen
- Inkrementelle Änderungen
  - Man kann Methoden und Datenfelder hinzufügen…
  - …aber nicht entfernen
    - Sonst hätten wir ein „Loch im Typsystem“

Andere Programmiersprachen

C/C++

ADTs in C++

```
class Rectangle { //  „class“ = „struct“ mit private: als Voreinstellung

 private: //  Bei „class“ auch schon voreingestellt
   Vector2d pos;
   float width;
   float height;









 public:

   Rectangle(Vector2d p, float w, float h) { // Konstruktor
      pos = p; width = w; height = h; numRectsMade++;}

   void getWidth() {return width;}
   void setWidth(const float w) {width = w;}


};
```

ADTs in C++

```
class Rectangle { // „class“ = „struct“ mit private: als Voreinstellung

 private: // Bei „class“ auch schon voreingestellt
   Vector2d pos;
   float width;
   float height;


 protected:
   float getLowerRightCorner() {
      return Vector2d(pos.x + width, pos.y + height);
   }

 public:

   Rectangle(Vector2d p, float w, float h) { // Konstruktor
      pos = p; width = w; height = h; numRectsMade++;}

   void getWidth() {return width;}
   void setWidth(const float w) {width = w;}


};
```

ADTs in C++

```
class Rectangle { // „class“ = „struct“ mit private: als Voreinstellung

 private: // Bei „class“ auch schon voreingestellt
   Vector2d pos;
   float width;
   float height;
   static int numRectsMade = 0; // static = Nur einmal pro Klasse

 protected:
   float getLowerRightCorner() {
      return Vector2d(pos.x + width, pos.y + height);
   }

 public:

   Rectangle(Vector2d p, float w, float h) { // Konstruktor
      pos = p; width = w; height = h; numRectsMade++;}

   void getWidth() {return width;}
   void setWidth(const float w) {width = w;}

   static int getNumMade() {return numRectsMade;} // wie @staticmethod

};
```

Konstruktoren und Destruktoren

```
class FloatVector { // beliebig langer Vector von floats

 private: // Bei „class“ auch schon voreingestellt
   float* numbers = nullptr;
   int size = 0;

 public:

   FloatVector(int initialSize) { // Konstruktor
      size = initialSize;
      numbers = new int[initialSize];
   }

   void operator+(…) {…} // Zugriff (in C++ auch über Operatoren, ähnlich __add__)
   void operator[](…) {…}

   ~FloatVector() { // Neu: Destruktor! – sichereres Aufräumen
      delete[] numbers;
   }

};
```

Vererbung in C++

```
class Shape { // „class“ = „struct“ mit private: als Voreinstellung
 public:
   virtual void draw(QPainter *painter) = 0; // = 0 entspr. @abstractmethod
   virtual ~Shape() {} // Footgun: virtueller Destruktor muss in Oberklasse definiert werden
};

class Rectangle : public Shape { // public erben (alles andere macht kaum Sinn)
   Vector2d pos; float width; float height;
 public:
   virtual void draw(QPainter *painter) {
      painter->drawRect(pos.x, pos.y, width, heigth);
   }
};

class Circle : public Shape { // Erbt auch von Shape
   Vector2d center; float radius;
 public:
   virtual void draw(QPainter *painter) {…}
};
```

Nutzung polymorpher Subtypen

```
Shape* my_shape = nullptr; // Variable anlegen, unbedingt als Zeiger! (nicht erzwungen)

my_shape = new Rectangle(…); // Zuweisung Zeiger auf Subtyp ist erlaubt!
my_shape.draw(painter);      // klappt!
delete my_shape;             // Speicher freigeben (nicht vergessen)

my_shape = new Circle(…);    // Zuweisung Zeiger auf Subtyp ist erlaubt!
my_shape.draw(painter);      // klappt!
delete my_shape;             // Speicher freigeben (nicht vergessen)

Shape *make_some_shape() {return Rectangle(…);}
Rectangle* my_shape = make_shape(); // Typfehler vom Compiler! Shape zu Allgemein

Shape ohne_pointer = Rectangle(…);          // Erlaubt, aber tut nicht, was man denkt!
        // Unterklassen ohne Zeiger vermeiden (nur nutzen, wenn man genau weiß, was man tut)
```

Warum Zeiger?

Verschiedene Klassen

- Unterschiedlich groß
- Nachfahren brauchen u.U. mehr Speicher als Vorfahren
  - Auch in C++ werden alle Datenfelder vererbt (unlöschbar)
  - Durch Vererbung nur Datenfelder hinzufügbar
- Wertsemantik
  - Ohne Zeiger ist nicht „genug Platz“
  - Aber alle Zeiger sind gleich groß
  - Polymorphie via Zeiger
- In Python/Scala/JAVA *genauso* (nur versteckt)

Empfehlungen für OO-Design

„Best Practices“ für Kapselung

Abstrakte Datentypen

Wann `private`?

- Orthodox: Alle Variablen („mutable state“) kapseln
  - Repräsentation kann geändert werden ohne die Schnittstelle zu ändern
    - Zumindest theoretisch („leaky abstractions“ – Schnittstelle oft nicht völlig von Implementation trennbar)
  - „Echte“ Variablen in „getter“ / „setter“ Methoden verpacken

```
class Date { // Beispiel in C++ (wg. Platz)
   private:
     int day; int month; int year;
   public:
     inline int getDay() {return day;}
     inline void setDay(int d) {day = d;}
};
```

Abstrakte Datentypen

Wann `private`?

- Praktische Probleme:
  - Manchmal hoher Aufwand (getter/setter Code) auch wenn Änderungen sehr unwahrscheinlich
  - Abwägung, wie wichtig Kapselung ist
- Alternative (z.B. in Python, C#, Delphi): „Properties“
  - Getter/setter können wie Variablen zugegriffen werden
  - Nachträgliche Kapslung ohne Änderung der Schnittstelle
  - Allerdings Neuübersetzuung (C#, Delphi) nötig

Beispiel: Properties in Python

```python
class Date:
    def __init__(self):
        self.day: int = 1
        self.month: int = 1
        self.year: int = 1970


# Hauptprogramm
x: Date = Date()
x.day = 28
print(x.day)
```

```python
class Date:
    def __init__(self):
        self.__day: int = 1
        self.__month: int = 1
        self.__year: int = 1970

    @property
    def day(self):
        return self.__day

    @day.setter
    def day(self, day):
        self.__day = day

    # …


# gleiches Hauptprogramm
x: Date = Date()
x.day = 28
print(x.day)
```

Abstrakte Datentypen

Wann `protected`?

- Protected: Zugriff aus Nachfahrenklassen gestattet
- Schnittstelle für Abgeleitete Klassen
  - Was braucht man (nur) um Nachfahren zu erstellen?
  - Methoden & Felder, die nur für „Ableiter“ wichtig sind
- Beispiel
  - Eventbehandlung in Widgets
    (z.B. „paintEvent“ zeichnet Widget neu; wird nicht von außen aufgerufen)
- Protected ist auch eine öffentliche Schnittstelle
  - Für Erweiterung durch Ableiten
  - Daher gut überlegen, Änderungen auch aufwendig

Abstrakte Datentypen

Wann `public`?

- Öffentliche Schnittstelle
  - Gut überlegen – Änderungen schwierig
- Implementationsdetails verstecken
  - Daher gibt es private
- Einfach verständlich, nicht zu kompliziert
  - Daher gibt es protected
  - Was „Endnutzer/in“ nicht braucht, wird versteckt

Objekt-Orientierter Entwurf

Kernkonzepte

Klassen

- Typen von Daten, die
  - Gleich strukturiert sind
  - Gleichartig verarbeitet werden
- **Objekte:** Instanzen der Klassen
- **Vererbung:** Beziehungen zwischen Typen

Methoden

- Code, der an Klassen gekoppelt ist
- Auswahl automatisch nach Typ (dynamic dispatch)

Entwurfsstrategie

Klassen finden

- Daten identifizieren
- Oft empfohlen:
  - Ausgangspunkt: Textuelle Problembeschreibung
  - Nomen finden → Nomen werden Klassen
  - Verben finden → Verben werden Methoden

Umstritten

- Kein Geheimrezept für tolles Design
  - Meine Meinung: Als Ausgangspunkt evtl. geeignet,
  - I.d.R. noch weitere Denkarbeit nötig

Beispiel

Beispieltext (Kundeninterview)

„Unsere Personalbuchhaltung verwaltet Daten von Personen. Für jede Angestellte werden zusätzlich Abteilung und Gehalt gespeichert. Die Angestellten können befördert werden, dabei erhöht sich das Gehalt. Monatlich fragen wir ab, wer bei uns angestellt ist.

Ach ja, und wir haben auch mehrere Managerinnen; die bekommen einen Bonus. Wenn das Gehalt berechnet wird, wird der Bonus aufgeschlagen.“

Beispiel

Analyse

- Klassen
  - Person
  - Angestellte
  - Managerin
  - (Bonus) → **`double`**
  - (Gehalt) → **`int`**
  - (Abteilung) → **`string`**
- Methoden
  - befördern()
  - berechneGehalt()
  - abfrage() → **`toString()`**

Weiteres Vorgehen

Ähnlichkeiten

- Vererbungshierarchien identifizieren
  - Ähnliches Verhalten (dies ist entscheident!)
  - Manchmal: Ähnliche Datenstrukturen (oft irreführend)
- Oberklassen formulieren
  - Abstrakte Operationen in Basisklasse
  - Implementationen / Spezialisierungen in Nachfahren
  - Gemeinsame Methoden in Basisklassen (DRY)

Kritik

Naives Modell

- „Nomen/Verb“-Feature ist instabil
- Kann in die Irre führen
  - Über-spezifische Designs
  - Natürliche Sprache ist oft wenig abstrakt
  - Datenverarbeitung / Code entscheidend, Namen egal

Vererbung: tiefgreifende Maßnahme

- Vererbung kann Designs unübersichtlich und schwer wartbar machen (Code über Hierarchie verteilt)
- Oberflächliche Ähnlichkeiten daher gefährlich

Vererbung?

Wann Vererbung benutzen?

- Vererbung ist eine „ist-ein“-Beziehung
- Im Code können alle Unterobjekte die Oberklasse ersetzen (und alles funktioniert wie intendiert)
- „Substitionsprinzip“
  (*Liskov substitution principle* nach Barbara Liskov)

Worauf achten?

- Verhalten!
  - Gleiche Methoden formen Oberklasse
  - Gleiche Invarianten, ähnliche Semantik

Sparsam Erben

„Anfängerfehler“ in OOP

- Zu viel Vererbung
- Gezielt einsetzen

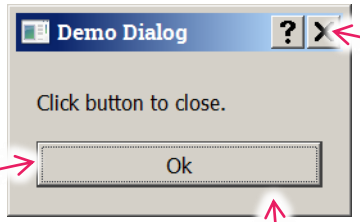
Keine Vererbung

- Delegation: Weiterreichen an andere Funktion
- Aggregation: Objekt besteht aus anderen Objekten

Ohne „ist-ein“: „Composition over Inheritance“

- In solchen Fällen: Objekte anderer Klassen als Datenfelder (Members) statt Oberklassen

Beispiel



Den Ausschalter „×“ vom Allgemeinen Button abstammen zu lassen, macht dagegen Sinn

Jedes Fenster hat genau einen „Ausschalter ×“ Button.

Fenster vom „Ausschalter“-Button abstammen zulassen ist aber eine sehr schlechte Idee.

Fenster „enthalten“ Ausschalter.

viele GUI Frameworks modellieren Buttons als spezielle Fenster, da die Funktionalität gleich ist (etwas unintuitiv, aber gut im Sinne des Substitutionsprinzips)

Wann Erben?

Interfaces / Subtyping für Polymorphie

- Verarbeitung von Datenobjekte
- Gleiche Operationen werden auf Objekte angewandt
- Oberklasse erlaubt, Algorithmus nur einmal zu schreiben

Wann Erben?

Interfaces vs. Oberklassen

- „Interfaces“: Nur abstrakte Methoden
  - Kein Code wird vererbt
  - Keine Datenstrukturen werden vererbt
- Oft gekoppelt an Mehrfachvererbung
  - Z.B. JAVA und SCALA erlauben nur eine Oberklasse, aber beliebig viele „Interfaces“ (JAVA) bzw. „Traits“ (Scala)
  - **Interfaces:** nur abstrakte Methoden
  - **Traits:** Zusätzlich mit Implementationen
  - Volle **Mehrfachvererbung:** Implementation + Daten
    - Problem, wenn Daten aus Basisklassen indirekt mehrmals geerbt werden

Wann Erben?

Braucht man volle Vererbung?

- Subtyping erfordert nur „Interfaces“
  - Schnittstellendefinition (ohne Code/Daten zu vererben) ausreichend für Typprürung
- In Python (u.ä.) Polymorphie ohne Subtypen möglich

Unterschied

- „Volle Vererbung“ stellt auch Code (+ Datenstrukturen) bereit
- Hierarchische Kompression des Codes

Wann Erben?

Erben von Code vs. reine Interfaces

- Code erben oder nur reine abstrakte Methoden?
- Nachteile von „voller“ Vererbung
  - Strukturelle Änderungen schwieriger
    - Klassen stärker gekoppelt
  - Hierarchische Struktur erforderlich
- Vorteil: „Frameworks“
  - Gemeinsame Funktionalität ist konsistent
  - Nur Details müssen ergänzt werden
    - Beispiel: Widget/GUI-Bibliotheken

Die übliche Antwort: „Es kommt drauf an…“

Wie entwirft man Oberklassen?

Oberklassen finden

- Gemeinsamkeiten extrahieren
  - Ziel 1: Polymorphe Algorithmen
  - ggf. Ziel 2: Code zusammenfassen

Oberklassen gestalten

- Fokus auf *Methoden*!
- „Was macht ein Objekt dieser Art?“
- Datenfelder möglichst „**`private`**“ machen und über Methoden zugreifen (→ Flexibilität)

Woran erkenne ich gutes Design?

Merkmale, die mit gutem Design korreliert sind

- „Single responsibility“
  - Jede Klasse hat (genau) einen, klar definierten Zweck
  - Jede Methode hat (genau) einen, klar definierten Zweck
- Orthogonalität
  - Minimale Wechselwirkung der Bestandteile
  - Freie Kombinierbarkeit (Lego®-artig)
- Konzeptionelle Klarheit
  - Einfach verständliches Interface
  - Offensichtliche Funktion, Annahmen, Invarianten
  - Wäre auch ohne Kommentare verständlich
    - Trotzdem gut dokumentiert

UML Diagramme

Entwurf von Software

Wie entwirft man Software?

- Diskussion am White-/Black-board
  - „Wandtafel“ in einem ruhigen Besprechungsraum oft wichtigstes Werkzeug
- Problem: Synchronisation der Vorstellungen
  - Jede(r) hat sein eigenes Gehirn
  - Bandbreite der Sprache gering
  - Vorstellungen liegen oft
    - dramatisch auseinander
    - unbemerkt!
  - Gemeinsame Sprache nötig

Werkzeug: Diagramme

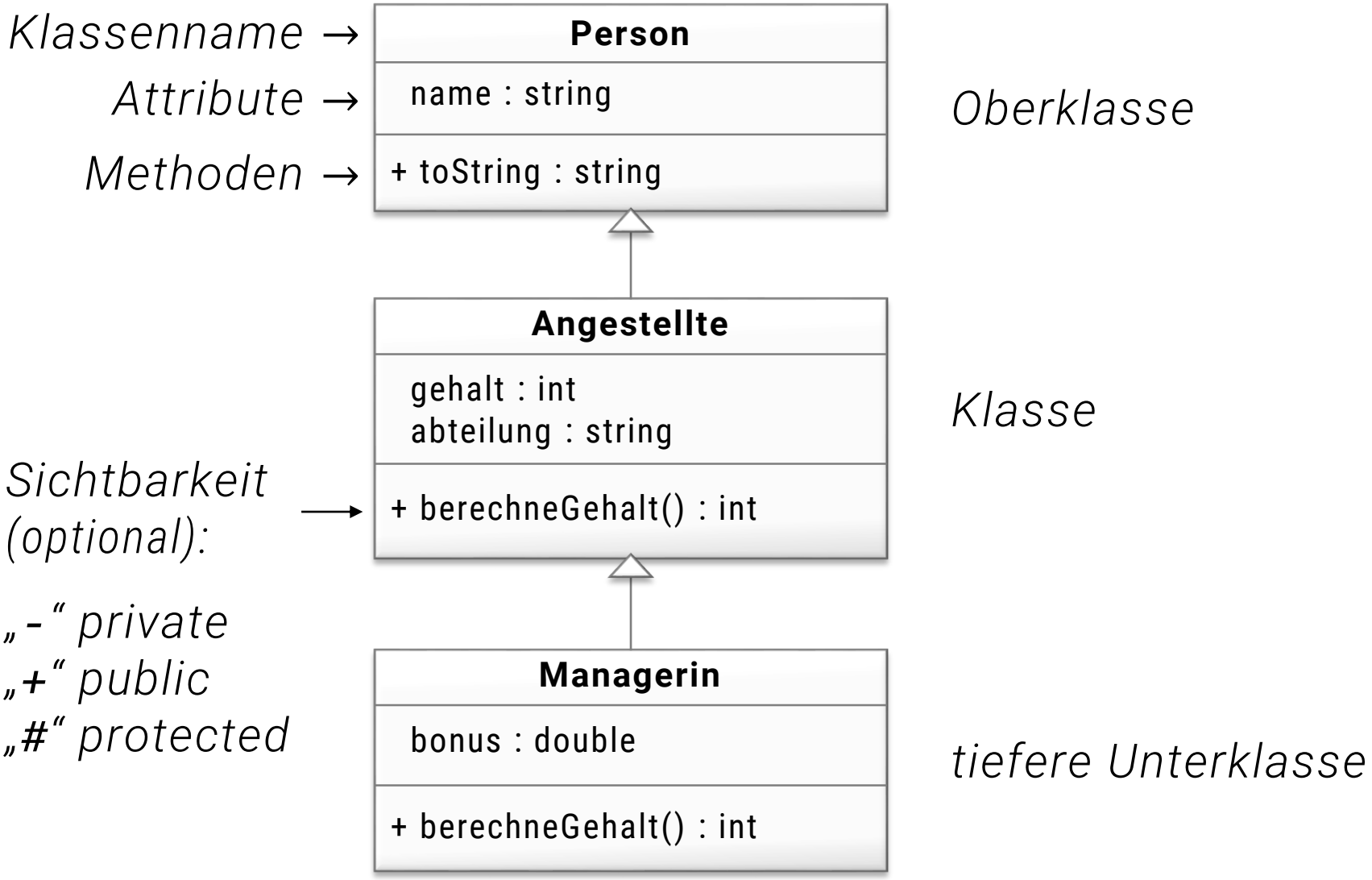
Hilfsmittel: Diagramme

- Visualisierung der *Struktur* und/oder des *Verhaltens* der Software
- Genormte Diagrammsprachen erleichtern klare Kommunikation

UML: „Unified Modeling Language“

- Definition einer Vielzahl von Diagrammtypen
- Recht genaue Definition der Semantik
- Fokus auf objektorientierte Entwürfe

Struktur: Klassendiagramme

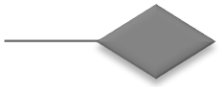


Beziehungen zwischen Klassen

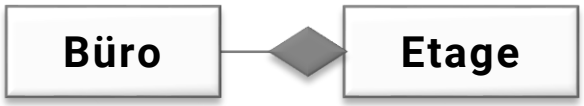
Vererbung
(Pfeil zeigt auf Oberklasse)

Assoziation
(Pfeil zeigt auf Klasse,
die referenziert wird)

Aggregation
(Raute an Klasse, die
das andere Objekt enhält)

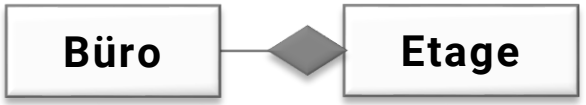


Komposition
(ausgefüllte Raute an Klasse,
die das andere Objekt enhält)



Unterschiedliche Zusammsetzung

Verschiedene Komposition



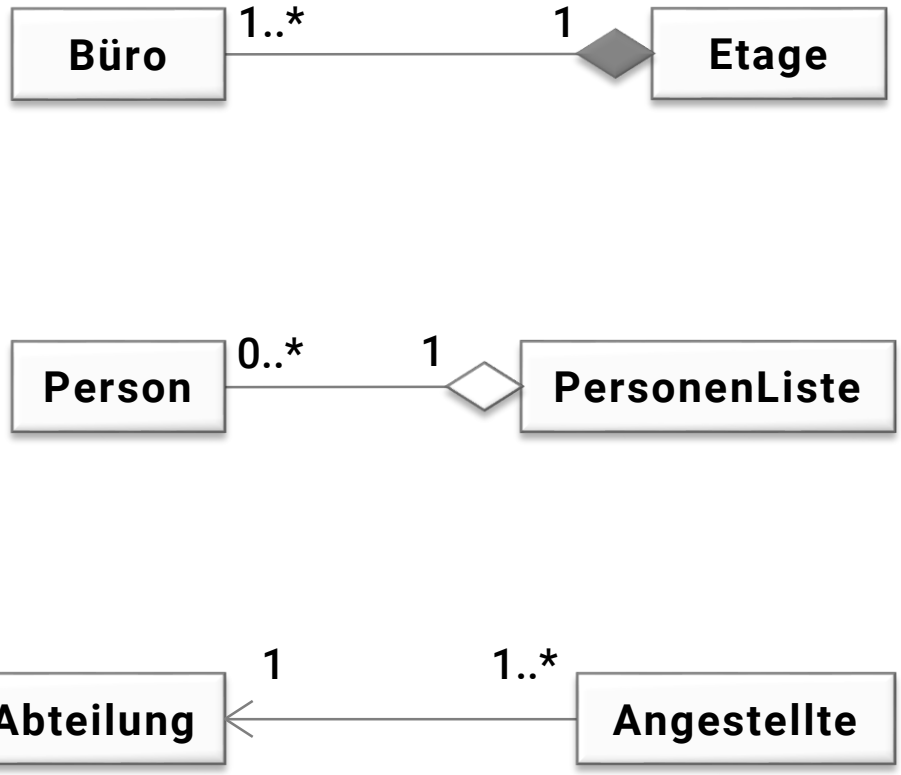
- „Komposition“
  - Klasse besteht aus Objekten anderer Klassen
  - Die Bestandteile müssen immer vorhanden sein

- „Aggregation“
  - Objekte enthalten Objekte anderer Klassen
  - Bestandteile brauchen nicht unbedingt vorhanden zu sein

- „Assoziation“
  - Objekte verweist auf Objekte anderer Klassen
  - Kein Bestandteil, nur eine Referenz

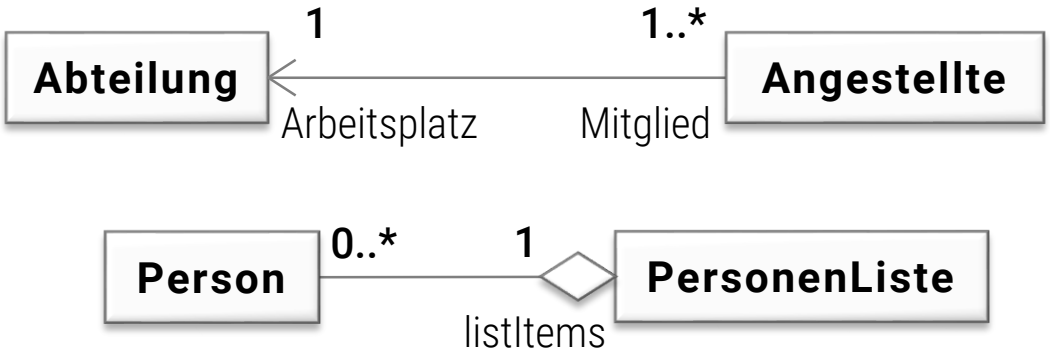
Multiplizitäten

Notation	
0	Keine Instanzen *(wenig Sinn)*
0..1	Keine oder eine Instanz
1	Genau eine Instanz
1..1	
0..*	Null oder mehr Instanzen
*	
1..*	Eine oder mehr Instanzen



Noch zwei Details

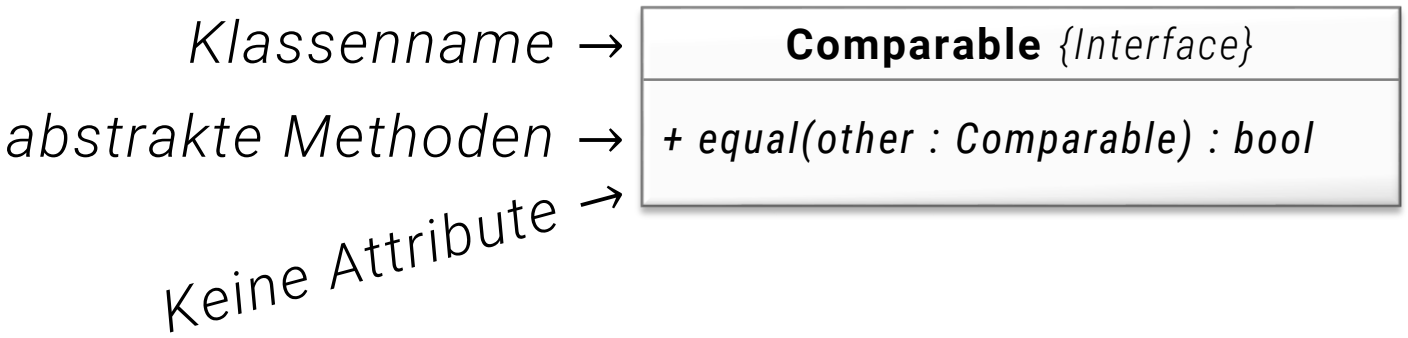
Rollenbezeichnungen
(optional)



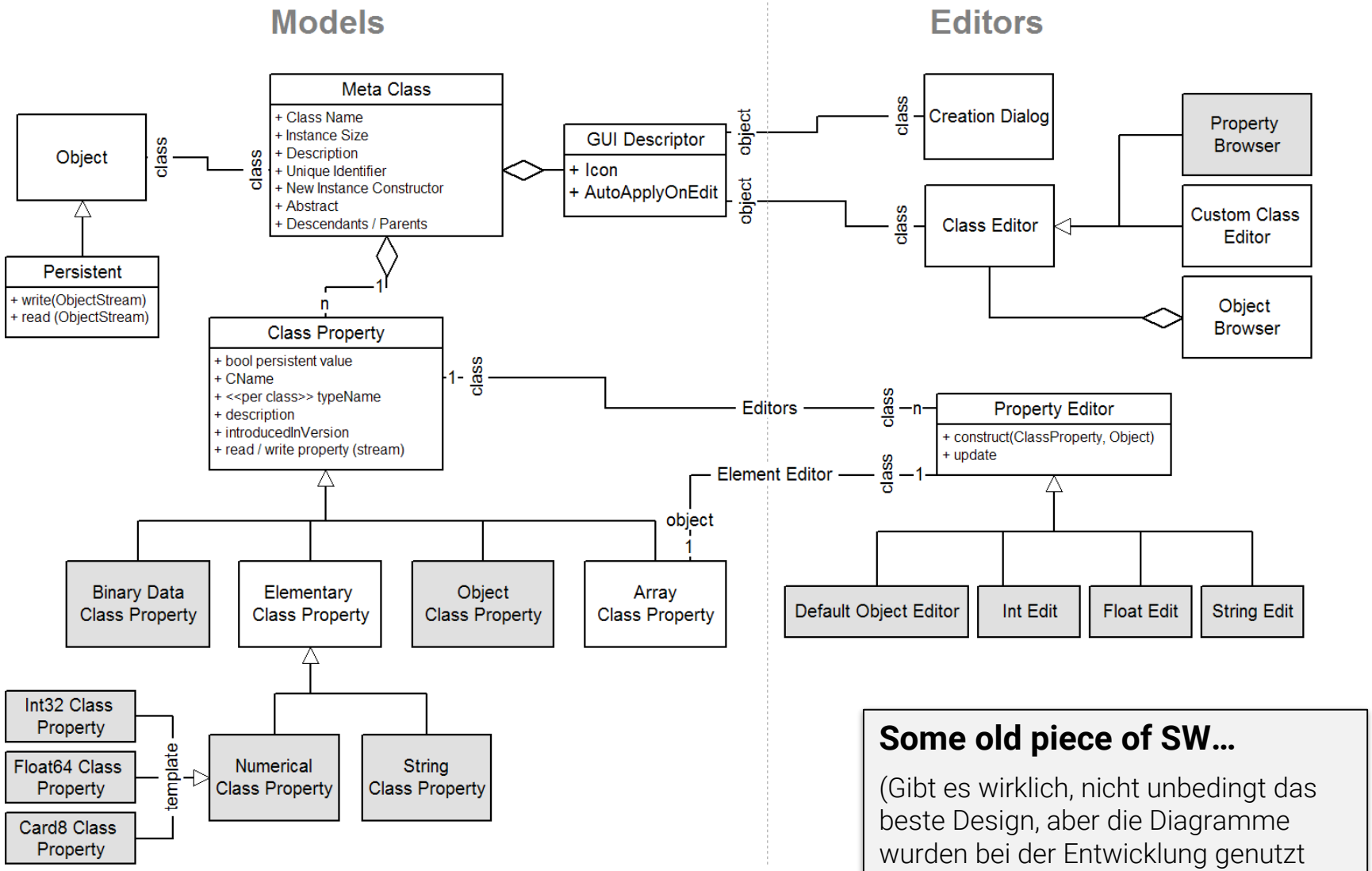
Noch zwei Details

Interfaces

(C++: Klassen nur mit abstrakten Methoden)



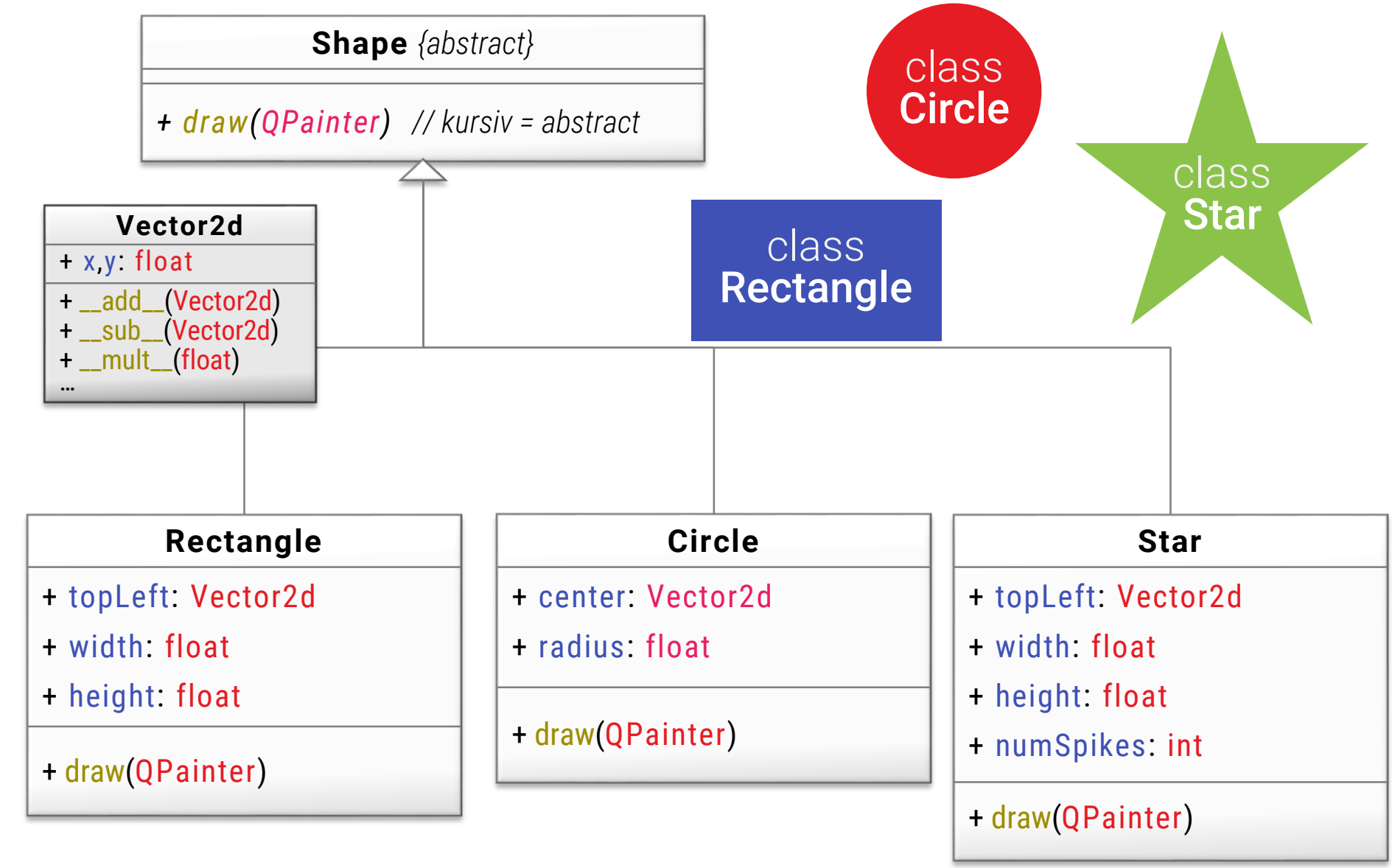
„Echtes“ Beispiel: GeoX Reflection System



Some old piece of SW…

(Gibt es wirklich, nicht unbedingt das beste Design, aber die Diagramme wurden bei der Entwicklung genutzt und so umgesetzt.)

Beispielentwurf EIS-Zeichenprogram


Shape {abstract}
+ draw(QPainter) // kursiv = abstract
class Circle
class Star
Vector2d
+ x,y: float
+ __add__(Vector2d)
+ __sub__(Vector2d)
+ __mult__(float)
...
class Rectangle
Rectangle
+ topLeft: Vector2d
+ width: float
+ height: float
+ draw(QPainter)
Circle
+ center: Vector2d
+ radius: float
+ draw(QPainter)
Star
+ topLeft: Vector2d
+ width: float
+ height: float
+ numSpikes: int
+ draw(QPainter)

Architektur

Shapes

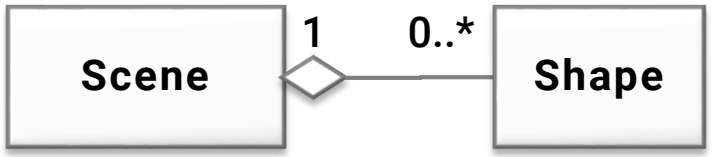
- Zeichnen sich selbst (erzeugen Pixelbild)
- Eigener Zeichenalgorithmus für jedes „Shape“

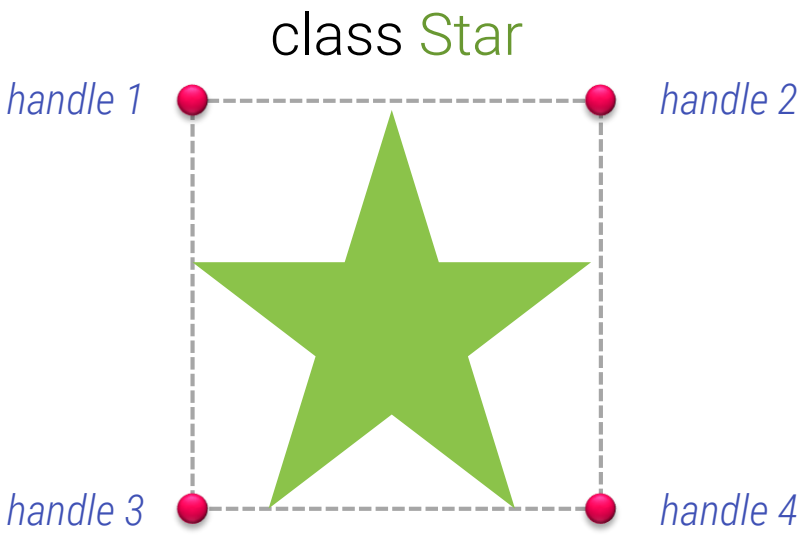
Nachfahren

- Implementieren das Zeichnen entsprechend
- Enthalten (zusätzliche) Attribute, die Form definieren

Zeichenalgorithmus

- `Scene` = Liste von „`Shape`s“
  - For each: `draw()`





Shape *{abstract}*

+ draw(QPainter) // Italics = abstract
+ getNumHandles() : int
+ getHandles(int) : Vector2d

Rectangle

+ topLeft: Vector2d
+ width: float
+ height: float

+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d

Circle

+ center: Vector2d
+ radius: float

+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d

Star

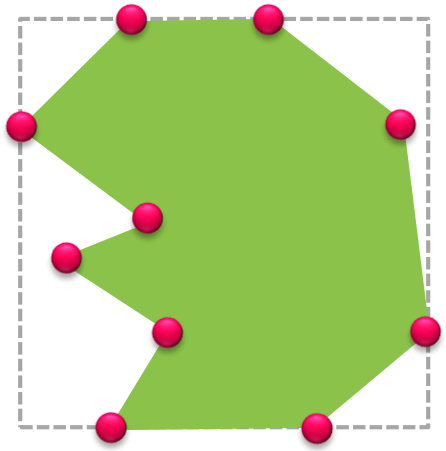
+ topLeft: Vector2d
+ width: float
+ height: float
+ numSpikes: int

+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d

Abstrakte Interaktion

Selektion von Objekten

- Handles holen
- Bounding-Box berechnen
- Testen, ob Mouse in BB fällt

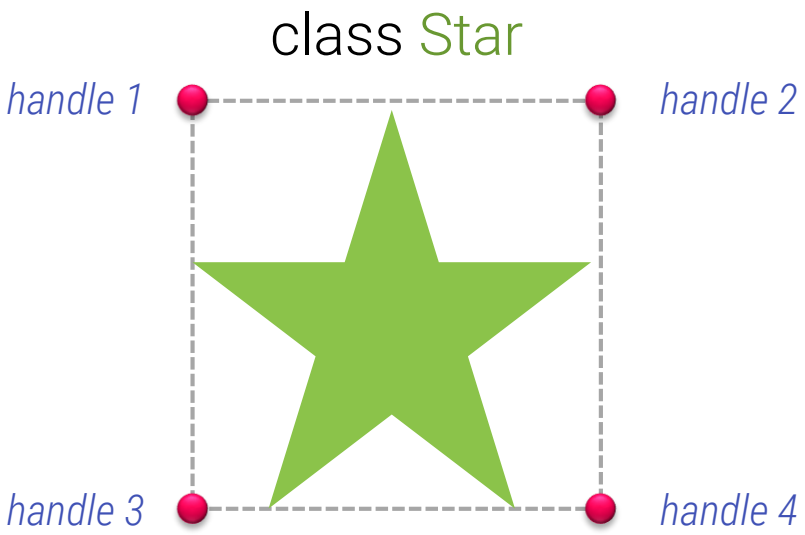


Polymorpher Algorithmus

- Für alle Objekte in Dokument
- Teste, ob angeklickt
  - Wenn ja, markiere als selektiert
- Für selektierte Objekte werden Handles gezeichnet

Beispiel-Code (grob)

```
#  find closest shape by handle (“picking”)
def pick_closest_handle(scene: Scene,                                   # ganze Szene
    world_x: float, world_y: float, world_width: float, world_height: float,  # Bildausschnitt Weltkoordinaten
    view_width: int, view_height: int,                  # …entspricht Bildausschnitt in Pixelkoordinaten
    mouse_x: int, mouse_y: int)                                         # Mauskoordinaten
-> Tuple[Shape, int] | None:        # Ergebnis: Referenz auf Objekt mit Index Handle oder None

    #  Konvertiere Pixel- (Maus-) Koordinaten zu Weltkoordinaten (view = Ausschnitt)
    mouse_x_world: float = mouse_x * world_width  / view_width  + world_x
    mouse_y_world: float = mouse_y * world_height  / view_height + world_y
    Vector2d mouseView = Vector2d (mouse_x_world, mouse_y_world)
    minDist: float = 1E20f // should be far enough…
    found: bool = False; shape: int = 0; handle: int = 0; minDist: float = 0
    for i in range(len(scene.shapes)):
        shape_ref: Shape  = scene.shapes[i]
        for j in range(shape_ref->getNumHandles()):
            handlePoint: Vector2d = shape_ref->getHandle(j)
            #  Distance = sqrt((x1 - x2)^2 + (y1 - y2)^2) (Pythagoras)
            dist: float = distance(handlePoint, mouseView)
            if (dist < minDist):
                found = True; shape = i; handle = j; minDist = dist
    if found:
        return scene.shapes[shape], handle
    else:
        return None
```



Shape *{abstract}*

+ draw(QPainter) // Italics = abstract
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)

Rectangle

+ topLeft: Vector2d
+ width: float
+ height: float

+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)

Circle

+ center: Vector2d
+ radius: float

+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)

Star

+ topLeft: Vector2d
+ width: float
+ height: float
+ numSpikes: int

+ draw(QPainter)
+ getNumHandles() : int
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)

Abstrakte Interaktion

Handles werden mit der Maus „verschoben“

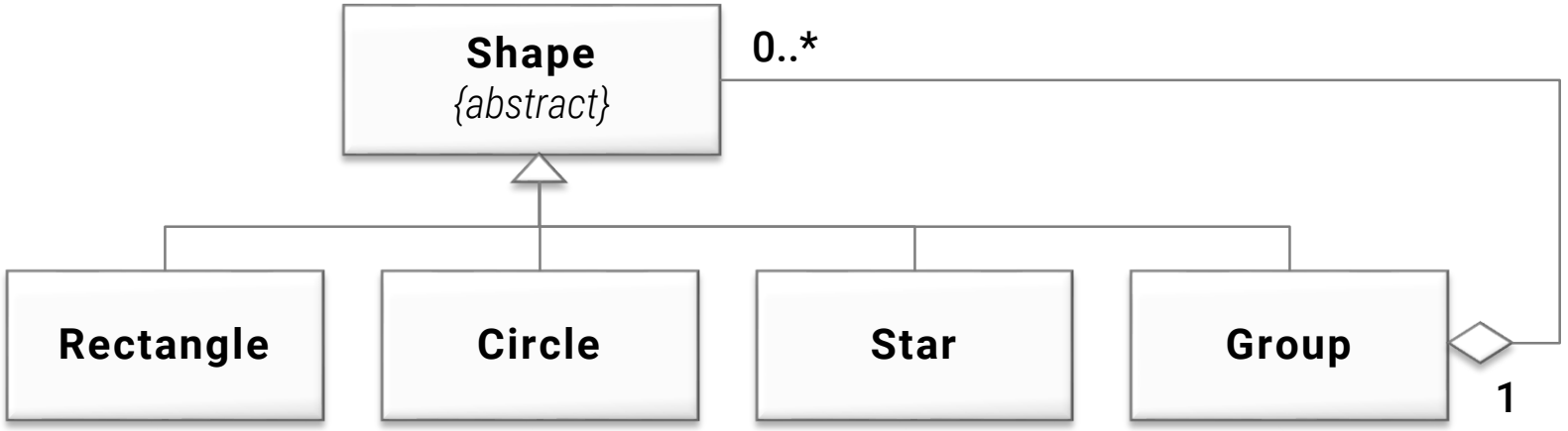
- Test für Anklicken gleich für alle Klassen
- Erkennung von „Mouse-Dragging“ gleich

Verschiebung

- Objekt mitteilen, dass „Handle“ sich verschoben hat
- Reaktion Klassenabhängig
  - Kreise bleiben immer rund
  - Rechtecke können sich rechteckig verzerren
  - Sterne: verschiedene Designs möglich

Nur ein Beispiel! (in der Übung besser machen :-))

Listen von Objekten (Gruppierung)



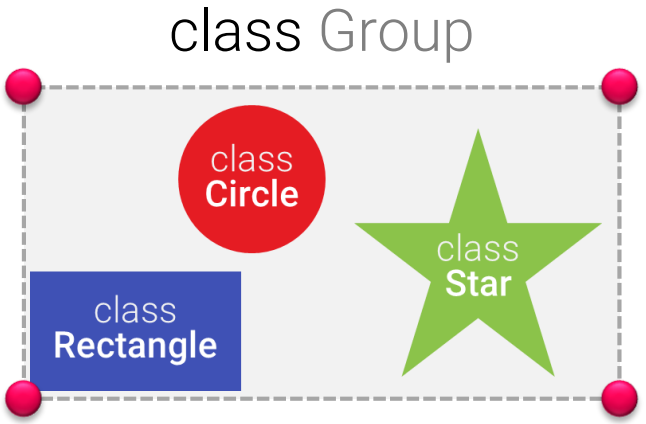
Indirekte Rekursion

Zeichenmethode

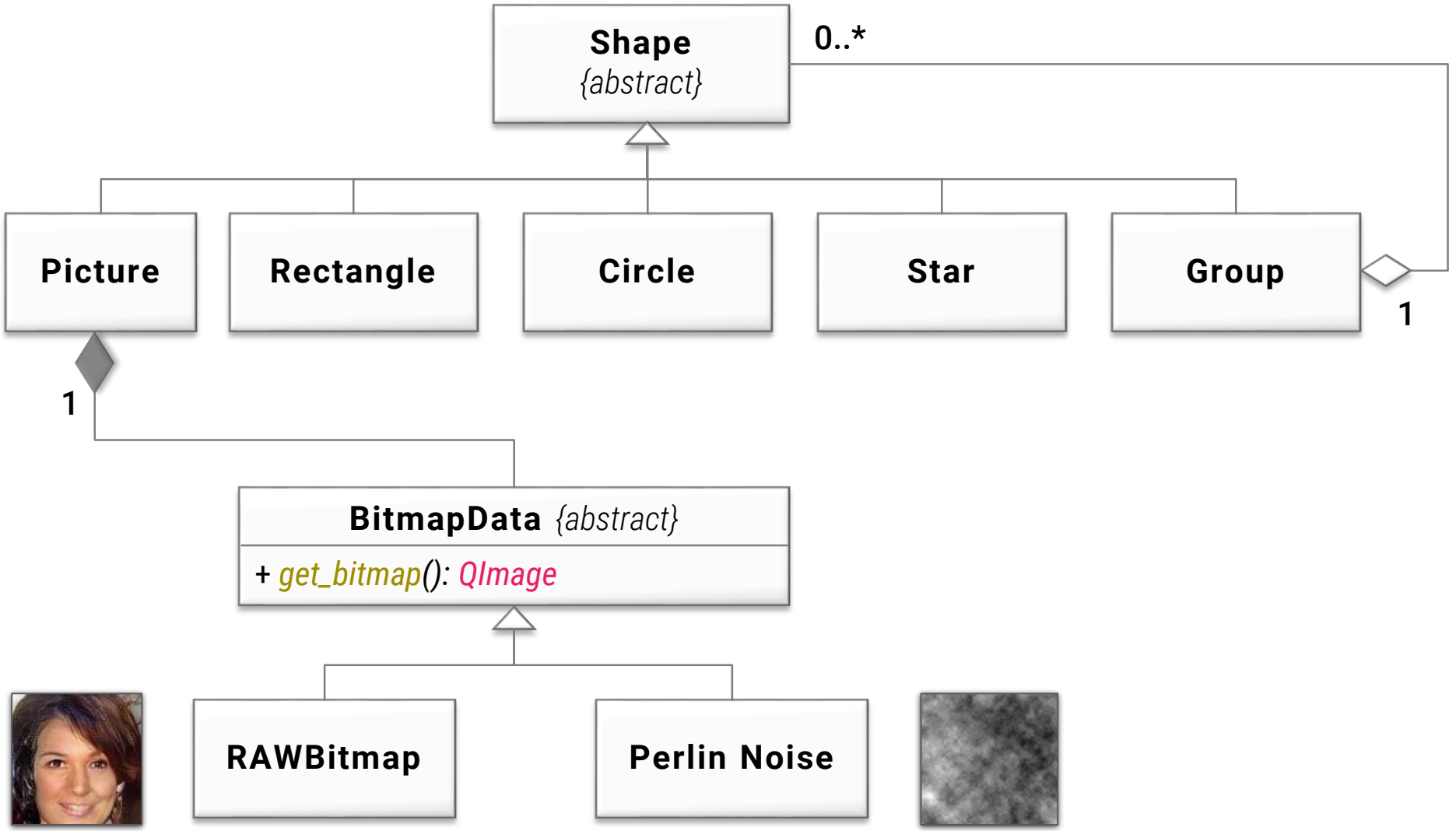
- Draw für „Group“ zeichnet alle enthaltenen Objekte
- Dies können auch Gruppen sein
- (Direkte und indirekte) Rekursion möglich

Handles für Gruppen

- Immer nur vier äußere Handles
- z.B. Bounding-Box aller Handles berechnen
- Transformationen weitergeben
  - Bounding-Box proportional verzerren



Bitmapbilder



loaded from PNG/JPG File

generated by noise algorithm

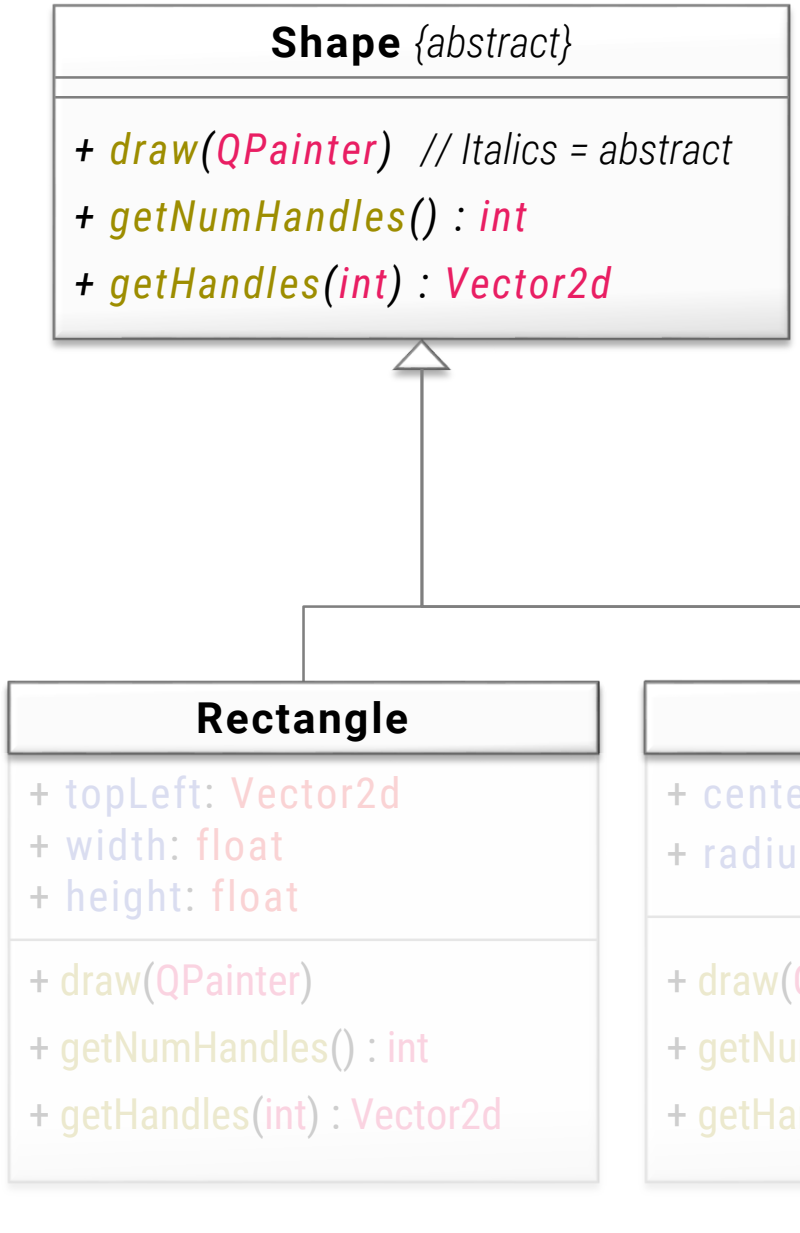
Was ist hier neu?

Designentscheidungen

- „Picture“-Shape getrennt von BitmapData
  - „Single Responsibility Prinziple“
  - Bitmaps auch außerhalb Zeichenprogramm nützlich / wiederverwendbar
  - Zu viel Vererbung kann schädlich sein
- Abstraktes Datenobjekt
  - BitmapData kann
    - …konkret sein: Pixeldaten im Speicher (z.B. aus PNG-Datei geladen)
    - …abstrakt sein: Daten werden „prozedural“ berechnet (Kein extra Speicherplatz nötig, flexible Änderungen)

Type-Checking

Subtyping



Sub-Types vs. Sum-Typs

- Shape $\equiv \sum$ alle Unterklassen
- Zusätzliche Garantie:
  Shape Interface vorhanden
- Erlaubt *typsichere* Polymorphie

Technischer Hintergrund

Wie ist das alles implementiert?

Zwei OOP Varianten

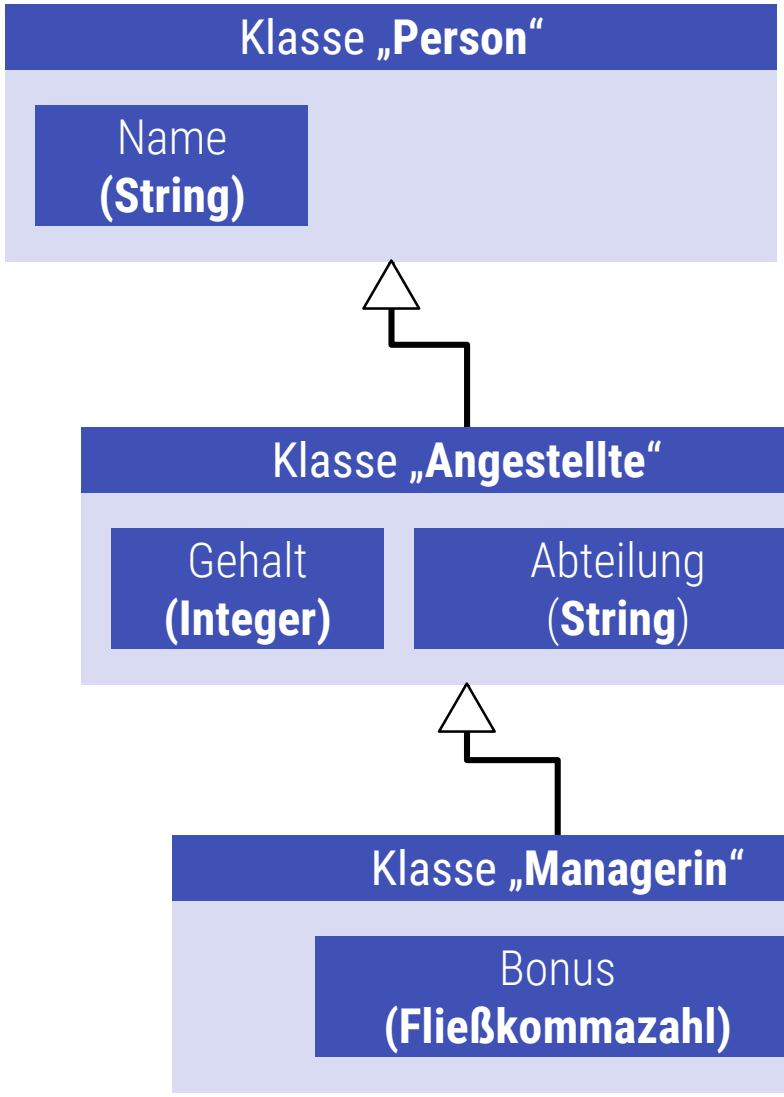
OOP in Python („Smalltalk“-Stil)

- Dynamische Typisierung via Ducktyping
  - Polymorphie durch „Nachrichten“ (dyn. Methodenaufruf)
- Alle Daten sind „Identitäts“-Objekte (Pointer)
- Sehr flexibel (alles dynamisch)

OOP in C++/Scala/Java/C# („Simula“-Stil)

- Statische Typisierung
  - Polymorphie durch „Subtyping“ via Vererbung
- Schneller & sicherer (Typprüfung)
- (Prinzipiell) Value- und Identity-Objects möglich

Beispiel: Datenmodellierung



Person: Eintrag für Personen in unseren Datenbestand

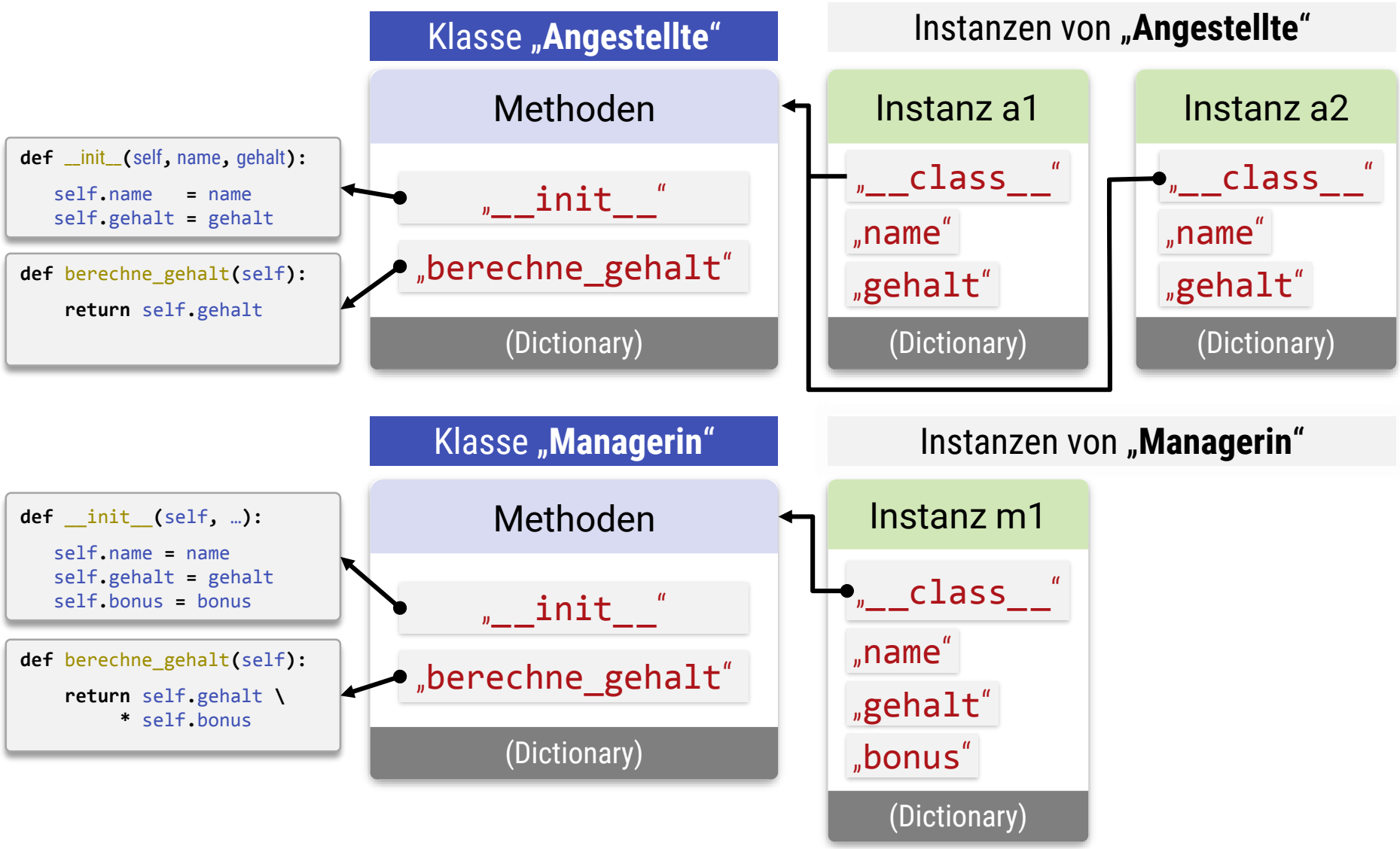
Angestellte: Arbeit in einer Abteilung, bezieht festes Gehalt

Managerin: Leistungsabhängiges Gehalt; Bonus ist ein Faktor zwischen [1.0,...,2.0]

Beispiel aus EiP: Vererbung in Python

```
class Person:
    def __init__(self, name):
        self.name = name
    def get_name(self):
        return self.name


class Angestellte(Person):
    def __init__(self, name, gehalt):
        super().__init__(name)
        self.gehalt = gehalt
    def berechne_gehalt(self):
        return gehalt

class Managerin(Angestelle):
    def __init__(self, name, gehalt, bonus = 1.1):
        super().__init__(name, adresse)
        self.bonus = bonus
    def berechne_gehalt(self):
        return int(super().berechne_gehalt()*bonus)
```

Implementation: Dictionaries



Python: Sehr einfach…

Implementation (im Wesentlichen)

- Klassen sind Sammlungen von Methoden
  - Dictionary: `Methodenname` → `Code`
- Objekte sind Sammlungen von Feldern
  - Dictionary: `Feldname` → `Objekt`
  - (andere Implementation möglich)

Implementationsdetails (noch mehr Flexibilität)

- Zugriff auf Felder über spezielle Methoden
  - *__getattr__*(), *__setattr__*() – Objekte ohne Dictionaries möglich (spez. Klasse)
  - Flexibilität : z.B. „immutable“ types
- Methodenaufruf auch programmierbar

In C++ / JAVA / Scala (u.ä.)

Beispiel in Python

```
class Person:
    def __init__(self, name):
        self.name = name
    def get_name(self):
        return self.name


class Angestellte(Person):
    def __init__(self, name, gehalt):
        super().__init__(name)
        self.gehalt = gehalt
    def berechne_gehalt(self):
        return gehalt

class Managerin(Angestelle):
    def __init__(self, name, gehalt, bonus = 1.1):
        super().__init__(name, adresse)
        self.bonus = bonus
    def berechne_gehalt(self):
        return int(super().berechne_gehalt()*bonus)
```

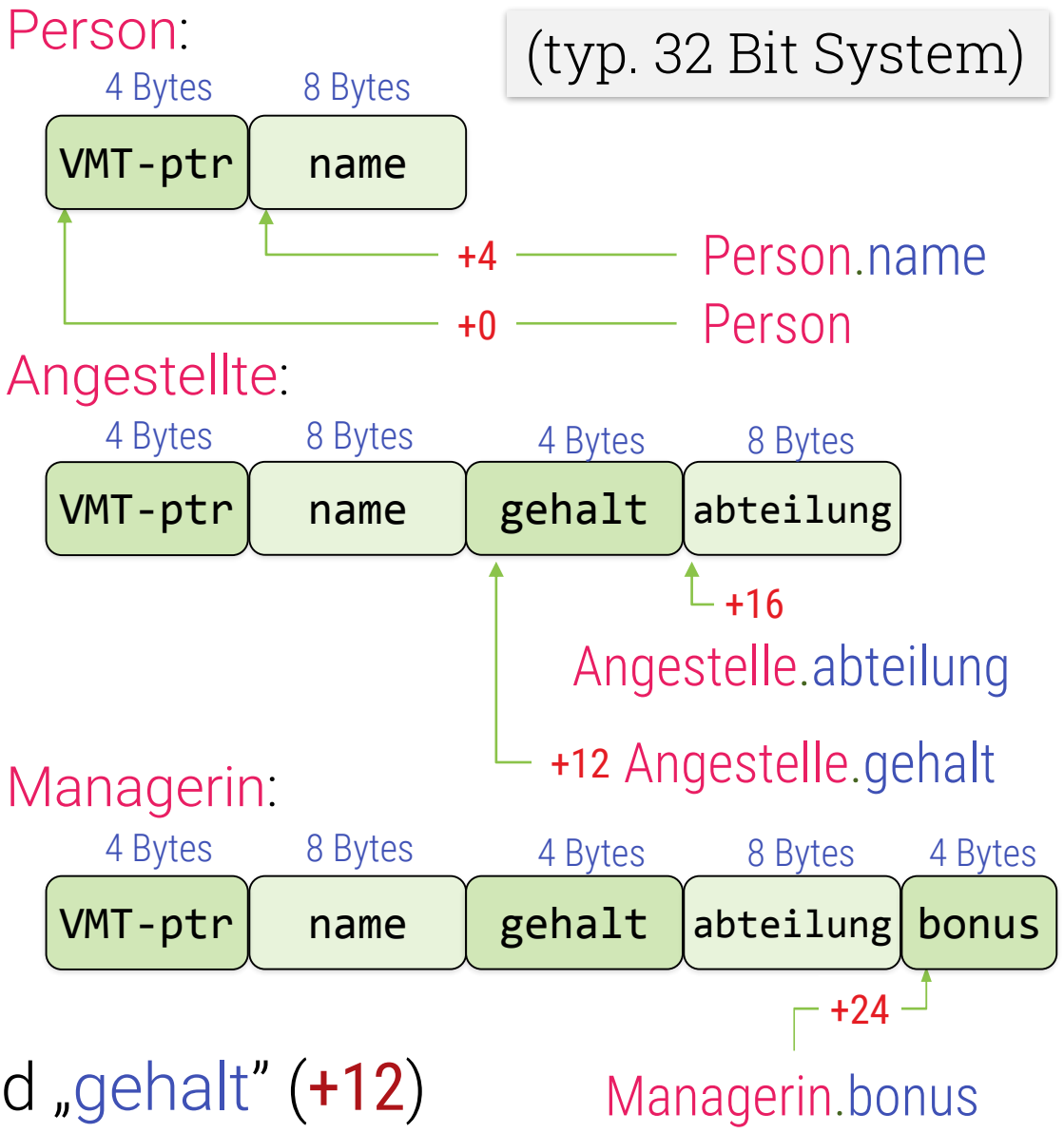
Beispiel in C++

```
class Person {
    Person(name: string) {
        this->name = name;    // impliziter Parameter this entspricht self
    }                         // Nützlich bei Mehrdeutigkeiten
    string get_name() {
        return name;
    }
};

class Angestellte: public Person {
    Angestellte(string name, int gehalt) : Person(name) {
        this->gehalt = gehalt;
    }
    int berechne_gehalt() {
        return gehalt;
    }
};

class Managerin: public Angestelle:
    Managerin(string name, int gehalt, float bonus = 1.1) : Angestellte(name, gehalt) {
        this->bonus = bonus;
    }
    int berechne_gehalt() {
        return int(Angestellte::berechne_gehalt()*bonus);
    }
};
```

Implementation in C++

Offsets

- Implementations-abhängig!
  - Hier nur das Prinzip!
- Offset 0: VMT Zeiger
- Offset 4: Person.name ≙ Zeiger + 4
  - Auch für Angestellte „Zeiger + 4“
- Angestellte: neues Feld „gehalt” (+12)

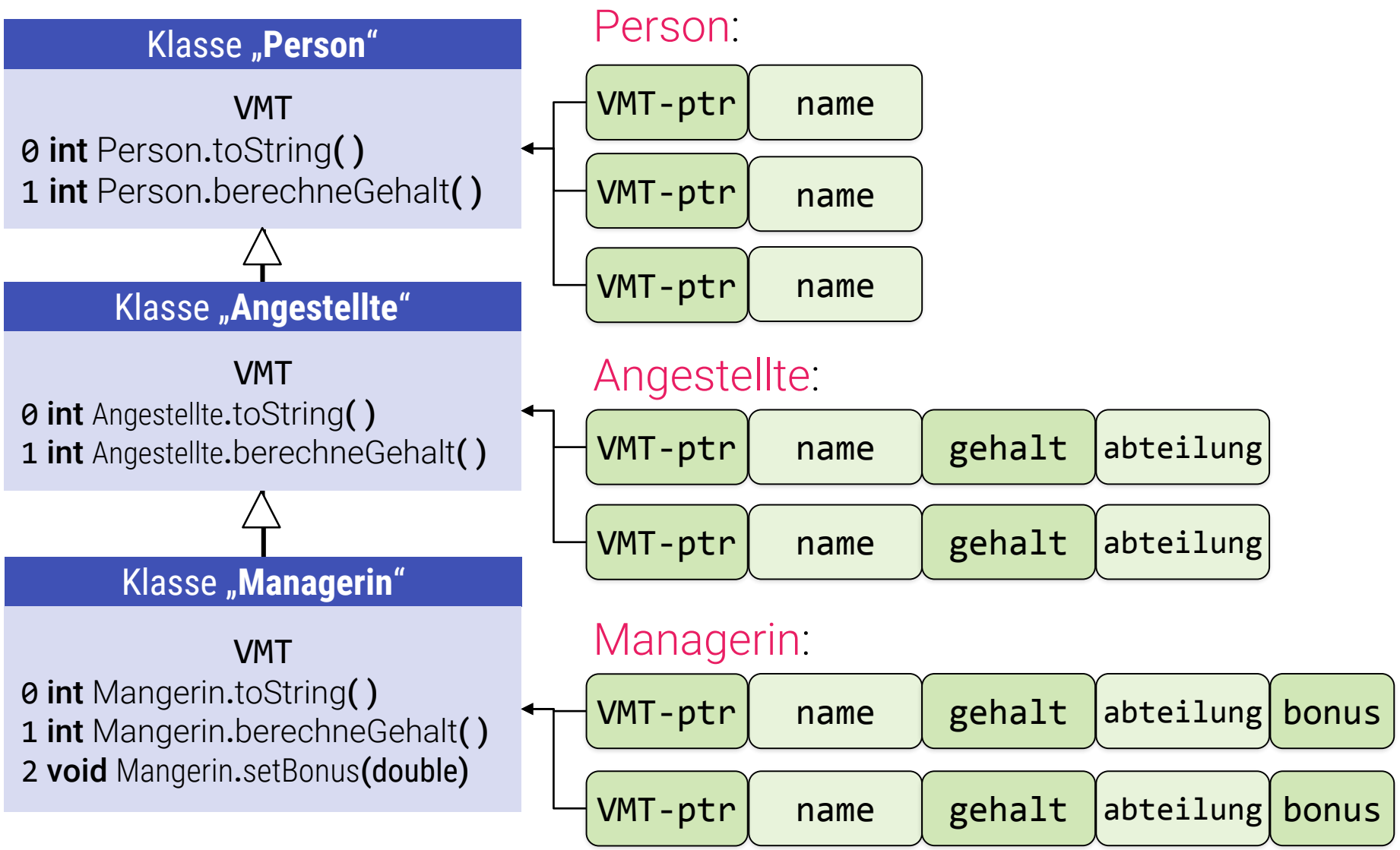


Virtuelle Methodentabellen

VMTs- Virtuelle Methoden Tabellen

- Englisch: „virtual method tables“
- Analog zu Python
  - Aber Offsets statt Hash-Tables *String → Function*
- Wesentlich schneller
- Statisches Subtyping nötig, um korrekte Funktion zu garantieren
- Weniger flexibel
  - Via Offsets ist (erstmal) kein Duck-Typing möglich!

Methoden: Dynamic Dispatch



Wie funktioniert das? – Emulation in reinem C

```
// Funktionszeiger – gibt es auch (genauso) in C++!
// Wir können annehmen, dass diese alle gleich groß sind (typ. 32/64-Bit Zeiger)
// Signatur ist nur Hinweis an Compiler für Aufruf; gespeichert wird immer das gleiche.
typedef void  (*Function)(void*);          // Pointer to simple function with pointer arg.
typedef char* (*ToStringFunction)(void*);  // Pointer to function returning char*
typedef int   (*BerGehaltFunction)(void*); // Pointer to function returning int

struct Person {
   Function *vmt; // C-Array! (Pointer to multiple functions)
   char *name;
};

struct Angestellte {
   Person parent; // Enthält VMT-Zeiger
   int gehalt;
};

char *Person_toString(void* self) {
   Person *typed_self = (Person*)self;
   return typed_self.name;
}
char *Angestellte_toString(void* self) {…}
int Angestellte_berechneGehalt(void* self) {…}
```

Wie funktioniert das? – Emulation in reinem C

```
const int PERSON_toString_INDEX 0 = 0

Person *Person_constructor() {
   Person *result = malloc(sizeof(Person));
   result.vmt = malloc(sizeof(Function));
   result.vmt[0] = (Function)&Person_toString();
   return result;
}

const int ANGESTELLTE_toString_INDEX = 0
const int ANGESTELLTE_berechneGehalt_INDEX = 1

Angestellte *Angestellte_constructor() {
   Angestellte *result = malloc(sizeof(Angestellte));
   result.vmt = malloc(sizeof(Function)*2);
   result.parent.vmt[0] = (Function)&Angestellte_toString();
   result.parent.vmt[1] = (Function)&Angestellte_toString();
   return result;
}

// Aufruf: Keine Typsicherheit in reinem C
Person *p1 = Angestellte_constructor();
p1->name = "Frau Meier";
char* result = ((ToStringFunction)p1.VMT[PERSON_toString_INDEX])(p1);
```