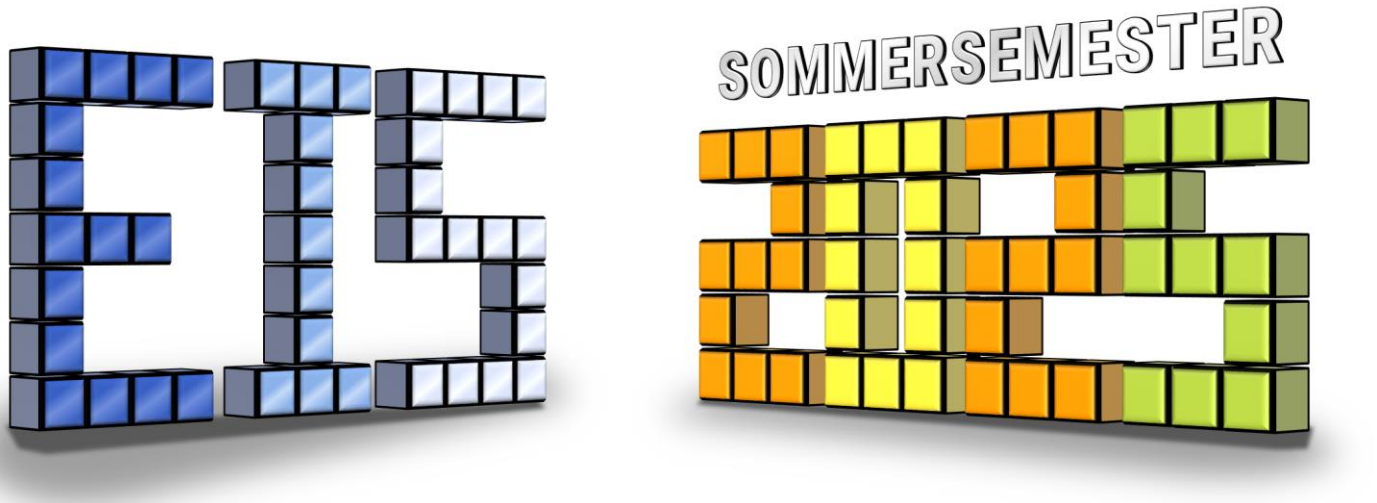
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #6

Prozedurale Programmierung

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

Techniken

Strukturierung von Programmen

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

Schauen wir uns mal das erste an…

Prozeduraler Entwurf

Prozedurale Programmierung

Prozedurale Programmierung

- Zwei Strukturelemente
  - Funktionen / Prozeduren
  - Datentypen
    - records (Pascal) / structs (C/C++) / „data classes“ (Python)

Zwei Fragen

- Entwurf der Datentypen
- Entwurf der Funktionen

Datentypen: Theorie

Theorie: drei Bausteine

- **Elementare Typen** sind Mengen von Werten:
  - z.B. Mengen A, B, C
- **Produkttypen**: Kartesische Produkte von Mengen
  - Records/Structs: $R = A \times B \times C$
  - Arrays:
    - Feste Länge: $V = A^n = \underbrace{A \times \cdots \times A}_{n-\text{mal}}, n \in \mathbb{N}$
    - Variable Länge: $V = \{\} \cup A^1 \cup A^2 \cup A^3 \cup \cdots$
- **Summentypen**: Vereinigung von Mengen
  - $U = A \cup B \cup C$
  - Arrays: variable Länge als „Summe“ von fixen Arrays

Produkttypen

Produkttypen

- $A \times B \times C$ für Mengen A, B, C
- Gemeint ist:
  - Eine Instanz von A,
  - kombiniert mit einer weiteren Instanz von B
  - kombiniert mit einer weiteren Instanz von C
- Mathematisches „Tupel“
  - Frei Kombination der Teile
  - Ein $a \in A$, ein $b \in B$, ein $c \in C$

    ergibt: $(a, b, c) \in A \times B \times C$

Praxis: Produkttypen in Python

Produkttypen in Python

Theorie

```
from dataclasses import dataclass

@dataclass
class Vector2d:
    x: float
    y: float

@dataclass
class Circle:
    center: Vector2d
    radius: float

my_circle: Circle = Circle(Vector2d(0.0, 0.0), 2.0)
my_circle.radius = 3.0
```

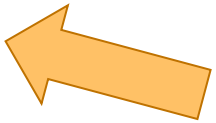
$$Vector2d = double \times double$$

$$Circle = Vector2d \times double$$

Praxis: Produkttypen in Python

`dataclass`: Spezialisierung von `class`

```
from dataclasses import dataclass

@dataclass
class Vector2d:
    x: float
    y: float
```



„Data Classes“ in Python:

- Erzeugt normale Python Klassen
- @dataclass „decorator“ erledigt einiges automatisch
  - Festes Datenlayout
  - **__init__** - Konstruktor mit Parametern (x, y)
  - Standardoperationen wie **__eq__**, Vergleiche, hashing…

Traditionelle Klassen in Python

Struktur definiert über „Konstruktor“ (→OOP)

```
class Vector2d:
    def __init__(self, x: float, y: float):
        self.x: float = x
        self.y: float = y

    #  optional: Weitere Operationen (z.B. auch __add__ etc.)
    def __eq__(self, other: Vector2d):
        return self.x == other.x \
               and self.y == other.y
```

…

```
example = Vector2d(23.0, 42.0)
```

Aufruf von `__init__`

Python: OOP Konzept (Konstruktor) unumgänglich

Dataclasses

Dataclasses

```
from dataclasses import dataclass

@dataclass
class Vector2d:
    x: float
    y: float
```

`@dataclass` erzeugt automatisch:

- `__init__(x: float, y: float)`
- Hilfsfunktionen (`__eq__`, `__repr__`, `__hash__`)
- Weniger Tipparbeit („Boilerplate Code“)

Produkttypen: Arrays

Python

```
// MyPy Typdefinition („from typing import TypeAlias, List“)
// Optional in Python
ListOfCircles: TypeAlias = List[Circle]

// Leere Liste erzeugen
my_list: ListOfCircles = []

// Ein Kreis hinzufügen
my_list.append(Circle(Vector2d(0, 0), 2))
```

Produkttypen: Arrays

Python

Theorie: $ListOfCircles = \bigcup_{k=0}^{\infty} Circle^k$

```
ListOfCircles: TypeAlias = List[Circle]
```

Produkttypen: Arrays

Python

Theorie: $ListOfCircles = \bigcup_{k=0}^{\infty} Circle^k$

```
ListOfCircles: TypeAlias = List[Circle]
```

Theorie: $List = \bigcup_{k=0}^{\infty} Any^k$

```
my_list: List = ["a", "b", "c", 23, 42]
```

Produkttypen: Arrays

Python

Theorie: $ListOfCircles = \bigcup_{k=0}^{\infty} Circle^k$

```
ListOfCircles: TypeAlias = List[Circle]
```

Theorie: $List = \bigcup_{k=0}^{\infty} Any^k$

```
my_list: List = ["a", "b", "c", 23, 42]
```

Theorie: $Tuple[float, float] = float \times float$

```
my_tuple: Tuple[float, float] = (23.0, 42.0)
```

In Programmiersprachen

Produkttypen

- Klassen/Structs/Records
  - Verschiedene Typen pro Eintrag
  - Benannte Einträge
- Arrays/Python „Listen“
  - Gleicher Typ für jedes Element (Python default: Any)
  - Ganzzahliger index benennt Eintrag
- Spezialfälle: Mischung aus beidem
  - Spezialfall: Unbenannte Structs, z.B. Python-„Tupel“
  - Zugriff hier per Index (außerdem „immutable“, feste Länge)

```
a: Tuple[int, str] = (23, "Hallo Welt")
print(a[0]) # ergibt 23
```

Andere Sprachen

Data-Classes in Scala

```
case class Vector2d(val x: float, val y: float);
case class Circle(val center: Vector2d,
                  val radius: float);
```

Case-Classes

- Entsprechen ungefähr den Dataclasses in Python
  - Auch eine Spezialisierung von „class“ (weniger allgemein)
  - Features wie Serialisierung & Hashing stehen automatisch bereit (Konstruktor gibt es ohnehin)
- Coding Guidelines
  - In der Regel „`val`“, also „immutable“ (oft Werte statt Objekte)
  - `val` und `var` möglich
    - Python Dataclasses sind mutable, Python „Tupel“ nicht

Scala

Arrays in Scala

- Feste Länge
  - `var myArray = new Array[Int](42);`
    - Leeres Array mit 42 Plätzen
- Veränderliche Länge
  - `var myVariableArray = new ArrayBuffer[Int]();`
    - Leeres Array mit flexibler Länge
    - In JAVA heißt das Pendent „ArrayList“

Praxis: Produkttypen in C

Produkttypen in C/C++

Theorie

```
struct Vector2d {
    double x;
    double y;
};
```

$$Vector2d = double \times double$$

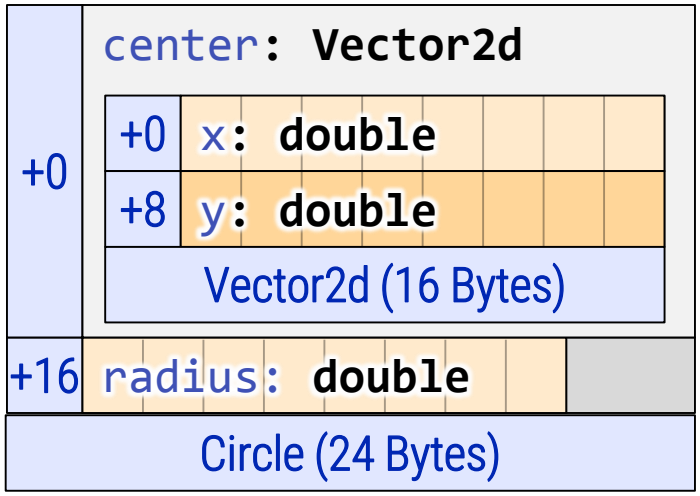
```
struct Circle {
    Vector2d center;
    double radius;
};
```

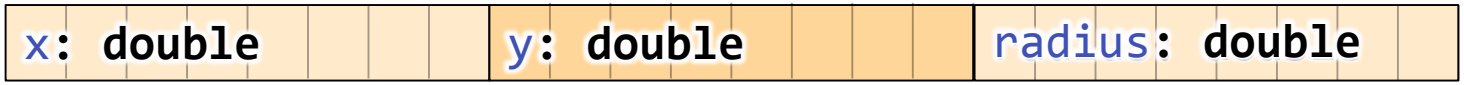
$$Circle = Vector2d \times double$$

Konstruktoren: Nur in C++ möglich, dort optional

Eigentlich nicht schwer…

```
struct Vector2d {
    double x;
    double y;
};

struct Circle {
    Vector2d center;
    double radius;
};
```





Ursprünglich (Algol, C, Pascal, etc…)

- „structs“: Datenfelder hintereinander im Speicher
- Typen für jedes Feld für Zugriff
  - Richtige Operationen für Zugriff
  - Offsets berechnen beim Zugriff

Das ist nicht schwer…

In C/C++ und Vorfahren (Pascal, Algol, etc.)

- Mehrere Datenfelder im Speicher
- Beim Array gleiche Einträge hintereinander
- Bei structs verschiedene Einträge hintereinander

In Python/Scala/Java etc.

- Komplexes Objekt-Modell im Hintergrund
- „einfache Structs“ muss man damit erst „emulieren“
- Ein Grund, warum wir auch C gelernt haben :-)

Produkttypen: Arrays

C/C++: Gewöhnungsbedürftige Notation…

…ansonsten auch simpel:

```
// Fixed lengths
// ListOf16Circles ist ein Array von 3 „Circle“s
typedef Circle ListOf3Circles[3];

// Variable lengths
typedef vector<Circle> ListOfCircles;

// Leere Listen anlegen
ListOf3Circles list = nullptr;
ListOfCircles variable_list;
```

Bausteine

Datentypen bauen

- **Elementare Typen**
  - `int`, `float`, `char`, (`string`) etc.
- **Produkttypen**
  - Records/Structs: $R = A \times B \times C$
  - Arrays:
    - Feste Länge: $V = A^n = \underbrace{A \times \cdots \times A}_{n-\text{mal}}, n \in \mathbb{N}$
    - Variable Länge: $V = \{\} \cup A^1 \cup A^2 \cup A^3 \cup \cdots$
- **Summentypen**: Vereinigung von Mengen
  - $U = A \cup B \cup C$
  - Arrays: variable Länge als „Summe“ von fixen Arrays

Summentypen

Einfache Idee…

- Type $T = A \cup B \cup C$ ist heißt:
  Variable vom Typ T ist **entweder**
  eine Instanz von A, **oder** von B, **oder** von C.
- $x \in T \quad \Rightarrow \quad x \in A$ oder $x \in B$ oder $x \in C$
- In der Praxis:
  - Wir müssen uns merken, was x ist (A/B/C)
  - Oft ignoriert man auch Überlapp
    - Annahme, das Schnittmenge immer leer ist
    - $A \cap B = B \cap C = A \cap C = \emptyset$
    - z.B. `int` und `float` seien disjunkt

Summentypen

Einfache Idee…

…komplexe Umsetzung (möglich)

- Einfach: Summentypen mit Pattern Matching
  - Wir schauen uns jetzt nur das an
- Komplexere Ideen
  - (Mehrfach-) Vererbung
  - Interfaces, Traits, Mixins
  - Parametrische Polymorphie mit Type-Constraints
  - Ziel: Gemeinsamkeiten festhalten für Polymorphie (Wiederverwendung von Code für verschiedene Datentypen)

Summentypen

Zwei Aspekte

- Implementation
  - Wie wird es realisiert?
- Typprüfung
  - Wie stelle ich sicher, das man nicht auf die falschen Typen zugreift?

Summentypen in Python

In Python

- Implementation
  - Trivial: Alle Objektreferenzen können alle Typen aufnehmen
  - (Technisch via Dictionaries realisiert)
- Typ-Prüfung
  - Wir müssen nur MyPy überzeugen, nicht Python!

Summentypen in Python

Summentypen in MyPy

- Einfaches Beispiel

```
a_number: float | int = 42
a_number = 23.0
```

- Alter Syntax (bis Python 3.9)

```
a_number: Union[float, int] = 42
a_number = 23.0
```

- Komplexerer Typ

```
Shape: TypeAlias = Circle | Triangle | Box
```

- MyPy prüft nun die korrekte Verwendung

Summentypen in Python

Typsicherer Zugriff (MyPy)

- Unterscheidung nach Typen bei Verwendung

```
def print_value(v: int | float):
    if isinstance(v, int):
        print(v, " (genau)")
    elif isinstance(v, float):
        print(v, " (ungefähr)")
    else: # alles andere…
        print("Das darf nie passieren!")
```

- Anmerkung: Unterscheidung float/int hier kosmetisch

Summentypen in Python

Typsicherer Zugriff (MyPy)

- Neuer Syntax (ab Python 3.10)

```
def print_value(v: int | float):
    match v:
        case int():
            print(v, " (genau)")
        case float():
            print(v, " (ungefähr)")
        case _: # alles andere…
            print("Das darf nie passieren")
```

Andere Sprachen...

In Scala

Sum Types + Patten Matching

```
case class Time(val hours:   Int,
                val minutes: Int);

case class Day(val day:   Int,
               val month: Int,
               val year:  Int);

type Data = Time | Day;

var y: Data = Time(23,59); // Deadline: 23h59!

val x: String = y match {
  case Time(h, m) => h.toString()+"h"+m.toString()+"m";
  case Day(d,m,y) => d.toString()+"."+m.toString()+"."+y.toString();
  case _ => "unknown";
}
```

Nice: Exhaustive Pattern matching
Scala gibt eine Warning, falls nicht alle Fälle berücksichtigt sind!

default case macht hier keinen Sinn (Scala warning), da alle Fälle vorhanden

Algol / Pascal / Modula-2

Algol, Pascal, etc.

- Discrimnated unions

```
TYPE DataType = (time, day);
TYPE Data = RECORD
  case theType: DataType of
    time: (hour: Integer; min: Integer);
    day: (d: Integer; m: Integer; y: Integer);
END;
```

- Sicherer Zugriff
- Kein exhaustivity-Check bei Benutzung

In C/C++?

In C

- Man bastelt sich das zusammen, z.B. mit Zeigern

```
struct Time{int m; int h;};

struct Day{int d; int y; int y;};

// das enum entspricht etwa: const int ADay=0; const int ATime=1;
enum DataType {ADay, ATime};

struct Data{
   DataType content;
   void *data;
};
```

- Es gibt auch „**union**“ („undiscriminated) in C/C++, aber das ist eine lange Geschichte…

In C/C++?

In C

- Benutzung dann etwa so:

```
enum DataType {ADay, ATime};
struct Data{
    DataType contentType;
    void *data;
};

Data x;
x.contentType = ATime;
x.data = new Time;
x.data->h = 23;
x.data->m = 59;
```

In C/C++?

In C

- Benutzung dann etwa so:

```
Data x;
x.contentType = ATime;
x.data = new Time;
x.data->h = 23;
x.data->m = 59;

if (x.contentType = ATime) {
    cout << ((Time*)x.data)->h
         << ((Time*)x.data)->m;
} else {
    …
}
```

„Harter“ Typecast:
Keine Prüfung durch Compiler!
Alles geht (auch Quatsch)

In C/C++

Schaut furchtbar aus

- Schon seit C gibt es „**`union`**“ – ohne Zeiger, aber auch ungeprüft
- In C++ würde man eher:
  - Eine Kapslung bauen
    - Geht auch für beliebige Summentypen via templates
  - Seit C++17 gibt es discriminated unions in der Standardbibliothek
    - Typ „variant“
  - Oder Vererbung nutzen
- Anders gesagt: Das Beispiel ist didaktisch gemeint :-)

Entwurfstechniken

Prozedurale Programmierung

„Strukturierte Analyse / Str. Design“ (1980er)

- Aufteilen des Programms in Funktionen
- Modellieren der Daten als Records & Arrays
- Visualisierung als *Datenflussdiagramme*

Analyse

- Modellierung z.B. von Geschäftsprozessen

Design

- Modellierung von Unterprogrammen

Datenflussdiagramme (DFDs)

Funktion

Input / Output

Datenspeicher

Datenfluss

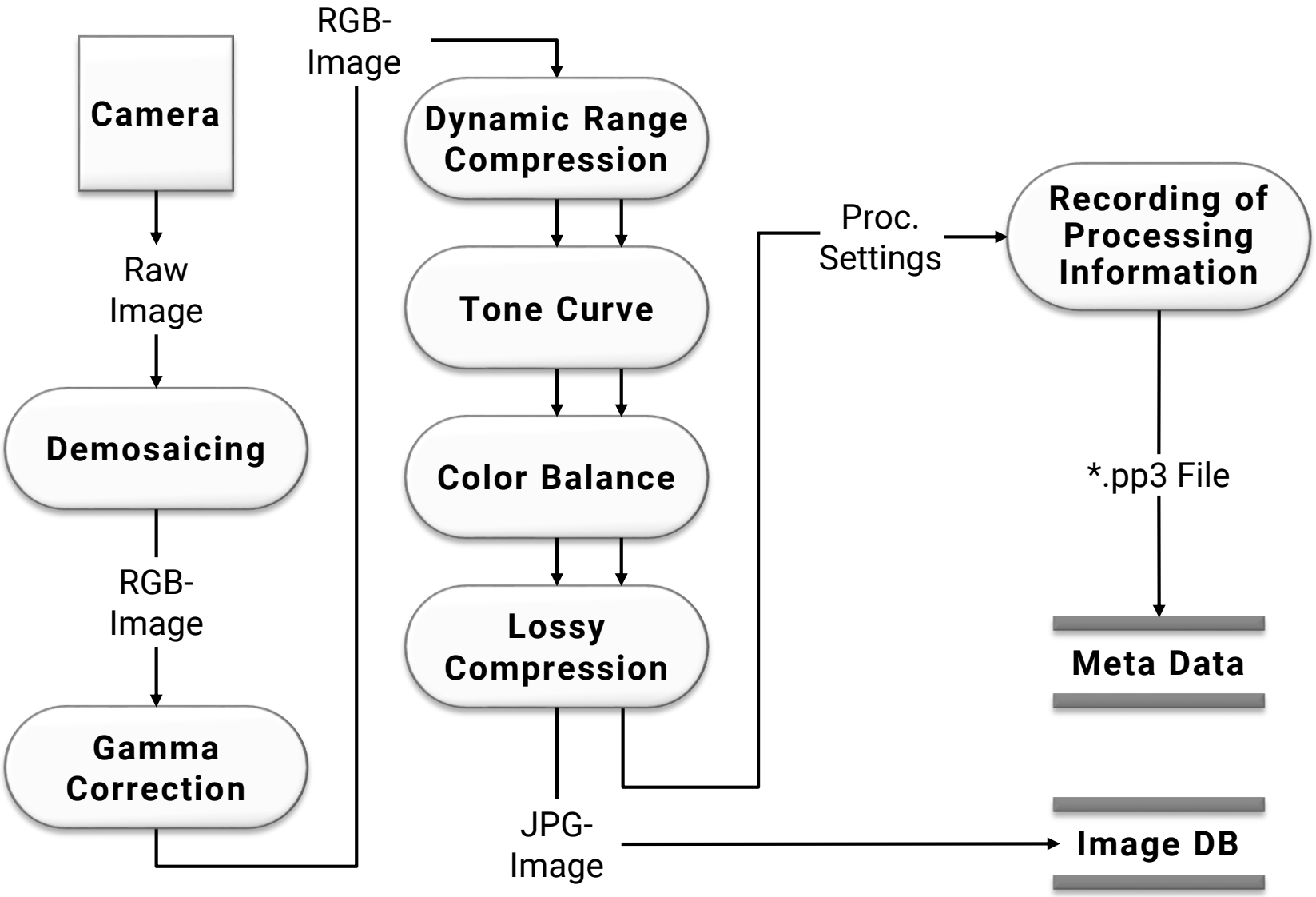
DFDs

- Alte Konvention (1980er)
- Hier genutzt, weil schön einfach :-)
- Nicht für Klausur auswendig lernen

Moderne Variante

- UML (Unified Modeling Language)
- „Activity Diagrams“

Beispiel DFD: RAW-Converter



Vorgehen

Wie gehe ich vor?

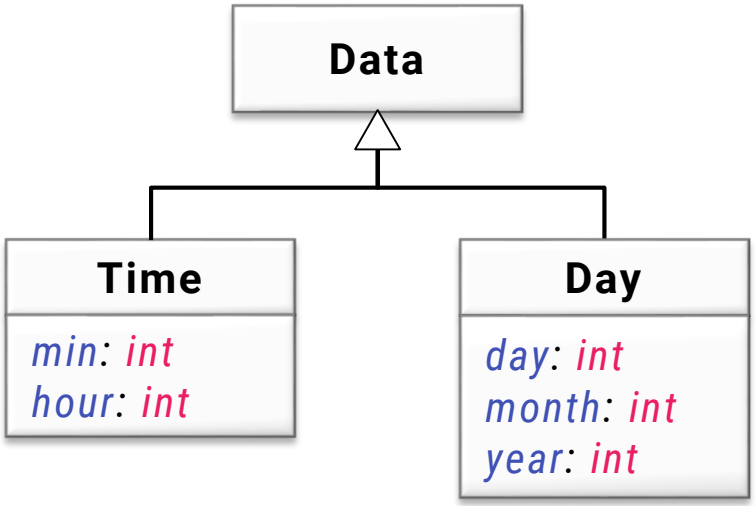
- „Divide and Conquer“
- Funktionen identifizieren
  - Vorgänge, Algorithmen
- Daten identifizieren
  - Welche Informationen fließen?
- Von grob nach fein
  - Immer weiter aufteilen
  - Bis Problem lösbar
- Bottom-up auch möglich
  - Bibliotheksdesign

Graphischer Entwurf von Datentypen

Ich nutze hier UML „Klassendiagramme“

- Vererbung vorerst als Summentypen interpretiert
  - (nicht ganz im Sinne der Erfinder von UML, aber ok)

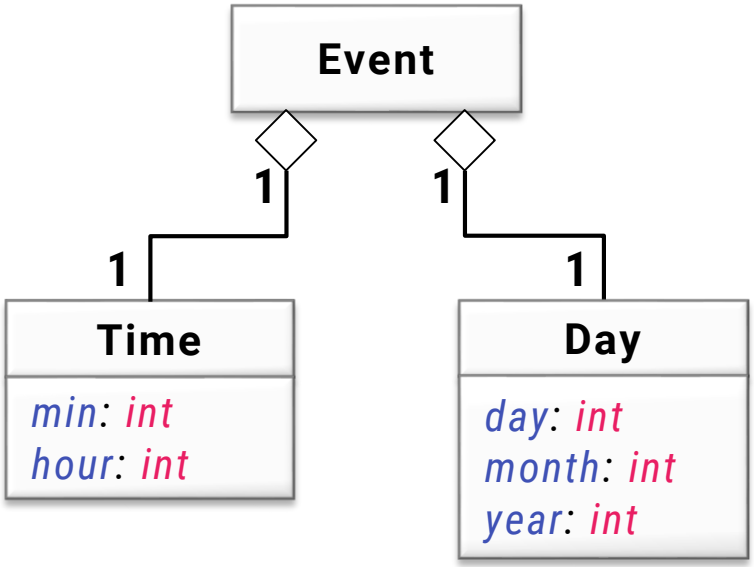
Summentypen



`Data` ist `Time` oder `Day`

Produkttypen

(Klassen, die Klassen enthalten)



`Event` enthält 1x `Time` und 1x `Day`

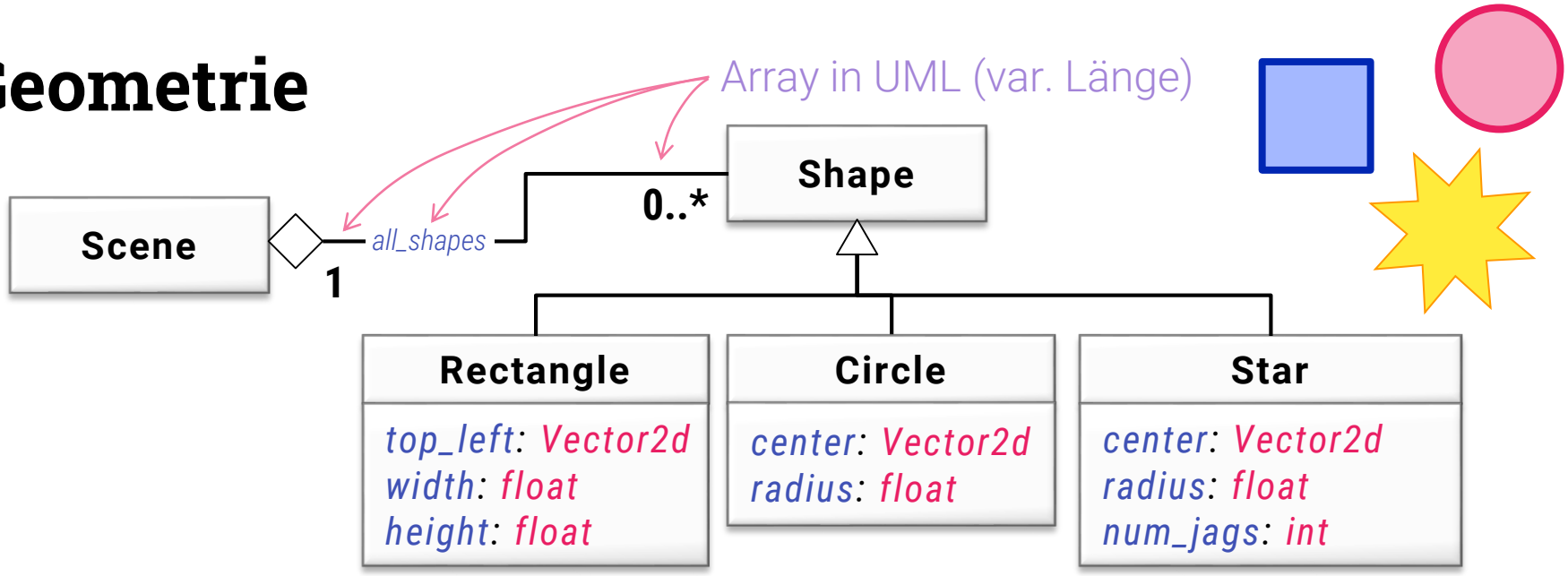
Entwurf Vektorgraphik-Bibliothek

Datentypen

Mathematische Typen (nur für Beispiel)

Vector2d	Matrix2x2	Transform2d
x: *float* *y*: *float*	*m11*, *m12*, *m21*, *m22*: *float*	*linear*: *Matrix2x2* *transl*: *Vector2d*

Geometrie



Code

```python
# Klassen für Geometrie
@dataclass
class Rectangle:
    pos: Vector2d
    width: float
    height: float

@dataclass
class Circle:
    center: Vector2d
    radius: float


@dataclass
class Star:
    center: Vector2d
    radius: float
    num_jags: int
```

```python
# Summentyp für alle Objekte
Shape: TypeAlias = Rectangle | Circle | Star


# gesammte Szene
@dataclass
class Scene:
    all_shapes: list[Shape]
```

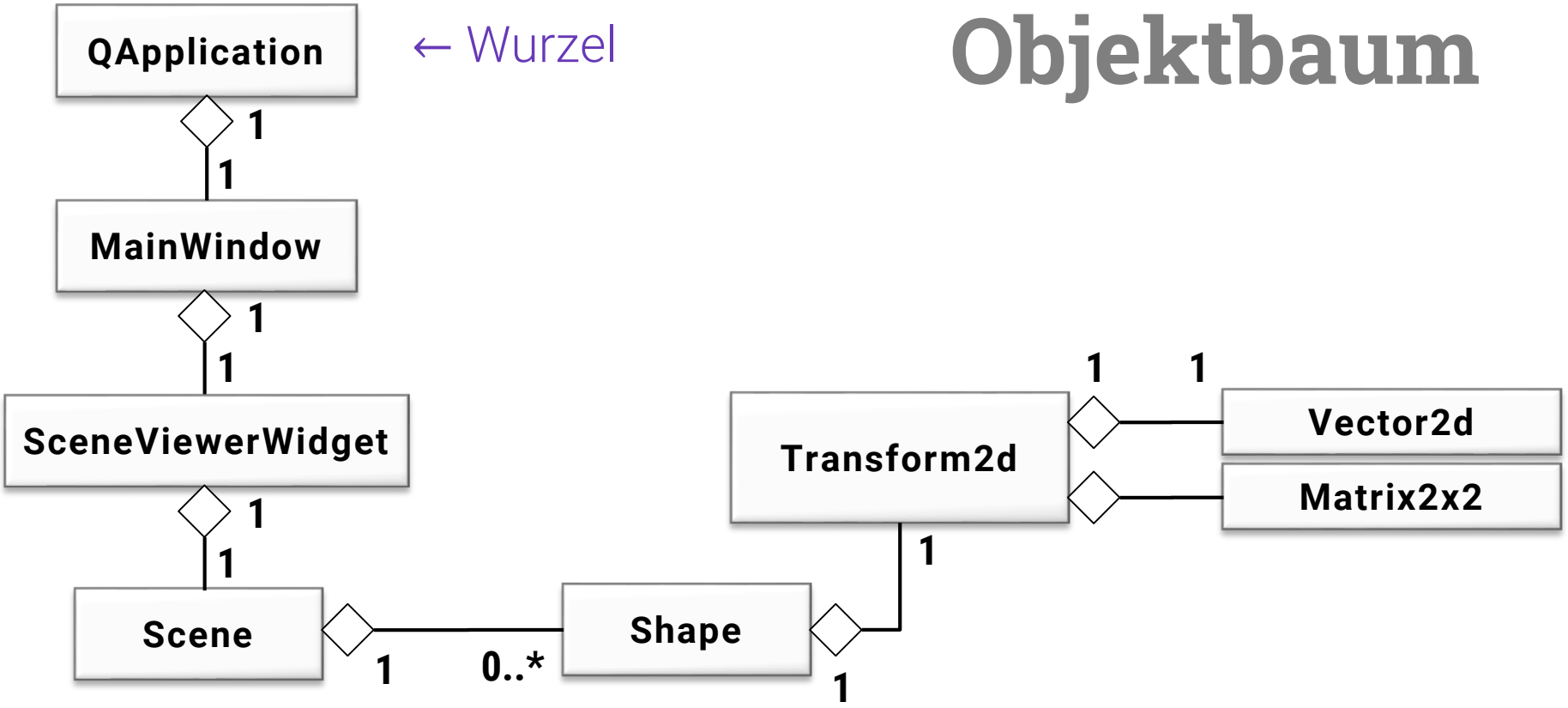
Alternativer Code

```python
# Klassen für Geometrie
@dataclass
class Rectangle:
    top_left: Vector2d
    bottom_right: Vector2d


@dataclass
class Circle:
    top_left: Vector2d
    diameter: float


@dataclass
class Star:
    top_left: Vector2d
    bottom_right: Vector2d
    num_jags: int
```

```python
# Summentyp für alle Objekte
Shape: TypeAlias = Rectangle | Circle | Star


# gesammte Szene
@dataclass
class Scene:
    all_shapes: list[Shape]
    file_name: str
    saved: bool
```

Entwurfs-Prinzip: „Bäume“ von Objekten

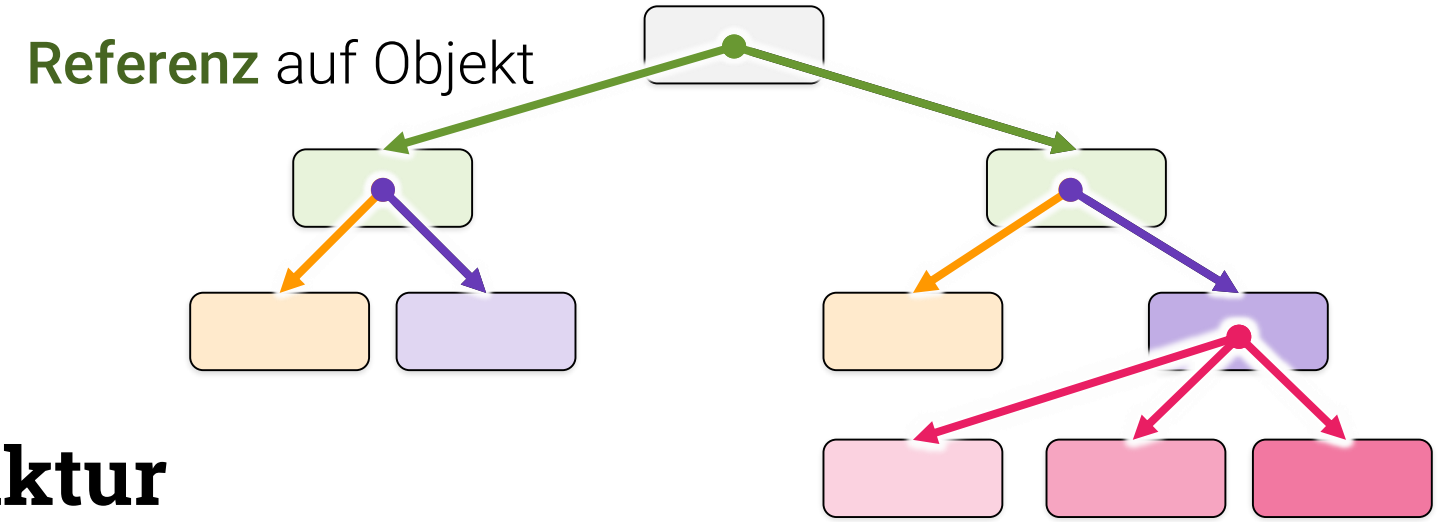
Objektbaum



Typischer Aufbau einer Anwendung

- Baum von Objekten, die einander enthalten
- Vereinfachtes Diagramm: nur Scene mit mehr als einem Kind
  - Tatsächlich enthalten andere Objekte auch viele Kinder

Baum von Objekten



Baumstruktur

- Geschachtelte Objekte

Ownership-Muster

- Besonders wichtig in C/C++ u.ä.
- Jedes Objekt hat einen eindeutigen „Besitzer“
  - zuständig für „`delete`“

Entwurfs-Prinzip: Abstrakte Datentypen

Abstrakte Datentypen

Beobachtung

- Verschiedene Optionen für Repräsentationen
- Repräsentation der Daten kann sich ändern

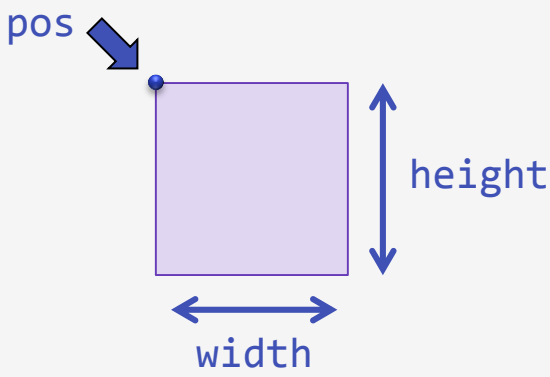
Flexibilität, Erweiterbarkeit

- Kein direkter Zugriff auf Daten!
- Alles per Unterprogramm
  - Unterprogramme kann man ändern
  - Der Rest des Codes bleibt gleich

„Abstrakter Datentyp“

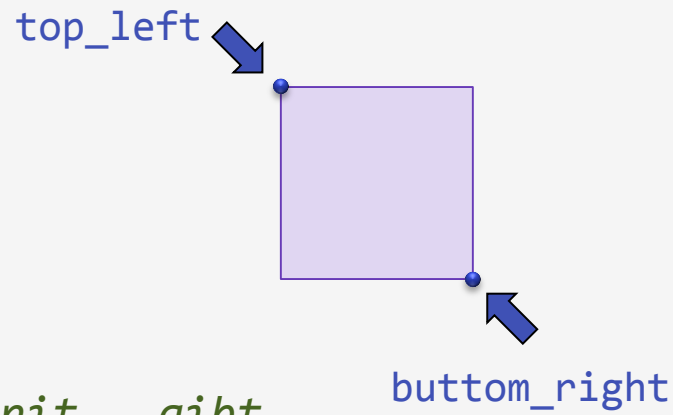
Beispiel: ADT

```python
@dataclass
class Rectangle:
    pos: Vector2d
    width: float
    height: float


# In Python redundant, da es schon __init__ gibt
# Konstruktoren sind das OOP-equivalent
def create_rectangle() -> Rectangle:
    return Rectangle(Vector2d(-0.5, -0.5), 1, 1)


# Implementation für „pos“ und „width“/„height“
def move_rectangle(vec: Vector2d, r: Rectangle) -> None:
    r.pos.x += vec.x
    r.pos.y += vec.y
```



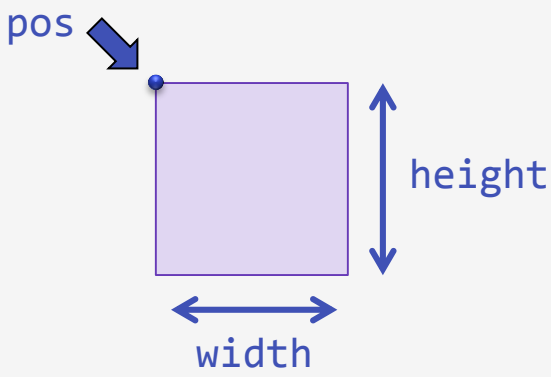
Beispiel: ADT

```python
@dataclass
class Rectangle:
    top_left: Vector2d
    bottom_right: Vector2d


# In Python redundant, da es schon __init__ gibt
# Konstruktoren sind das OOP-equivalent
def create_rectangle() -> Rectangle:
    return Rectangle(Vector2d(-0.5, -0.5), Vector2d(0.5, 0.5))


# Implementation für „top_left“ und „bottom_right“
def move_rectangle(vec: Vector2d, r: Rectangle) -> None:
    r.top_left.x += vec.x
    r.top_left.y += vec.y
    r.bottom_right.x += vec.x
    r.bottom_right.y += vec.y
```



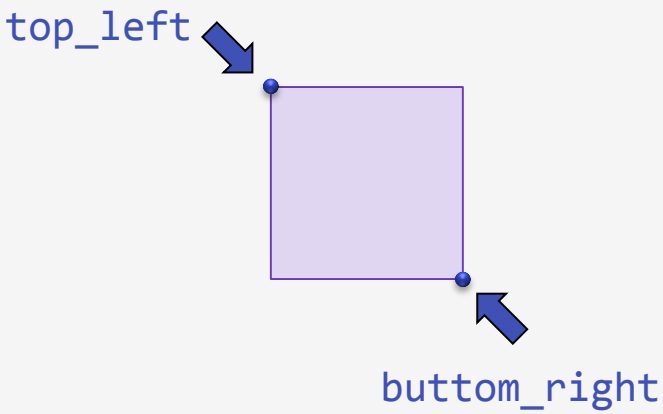
Beispiel: ADT

```python
@dataclass
class Rectangle:
    pos: Vector2d
    width: float
    height: float


# Berechnung Breite
def get_width(r: Rectangle) -> float:
    return r.width


# Berechnung rechte untere Ecke
def get_lower_right_corner(r: Rectangle) -> Vector2d:
    return Vector2d(r.pos.x + r.width, r.pos.y + r.height)
```



Beispiel: ADT

```python
@dataclass
class Rectangle:
    top_left: Vector2d
    bottom_right: Vector2d


# Berechnung Breite
def get_width(r: Rectangle) -> float:
    return r.top_left.x - r.bottom_right.x


# Berechnung rechte untere Ecke
def get_lower_right_corner(r: Rectangle) -> Vector2d:
    return r.bottom_right
```



Welche Operationen braucht man?

Allgemeine ADTs

- Minimum
  - Erzeugen von Objekten
  - ggf. Löschen von Objekten (vor allem ohne GC)
  - Setzen und Abfrage von Eigenschaften
    - Hier z.B. Breite, Höhe, Position
- Oft benötigt
  - Vergleich
  - Kopien erstellen
  - Serialisierung (z.B. Laden, Speichern; mehr dazu später)

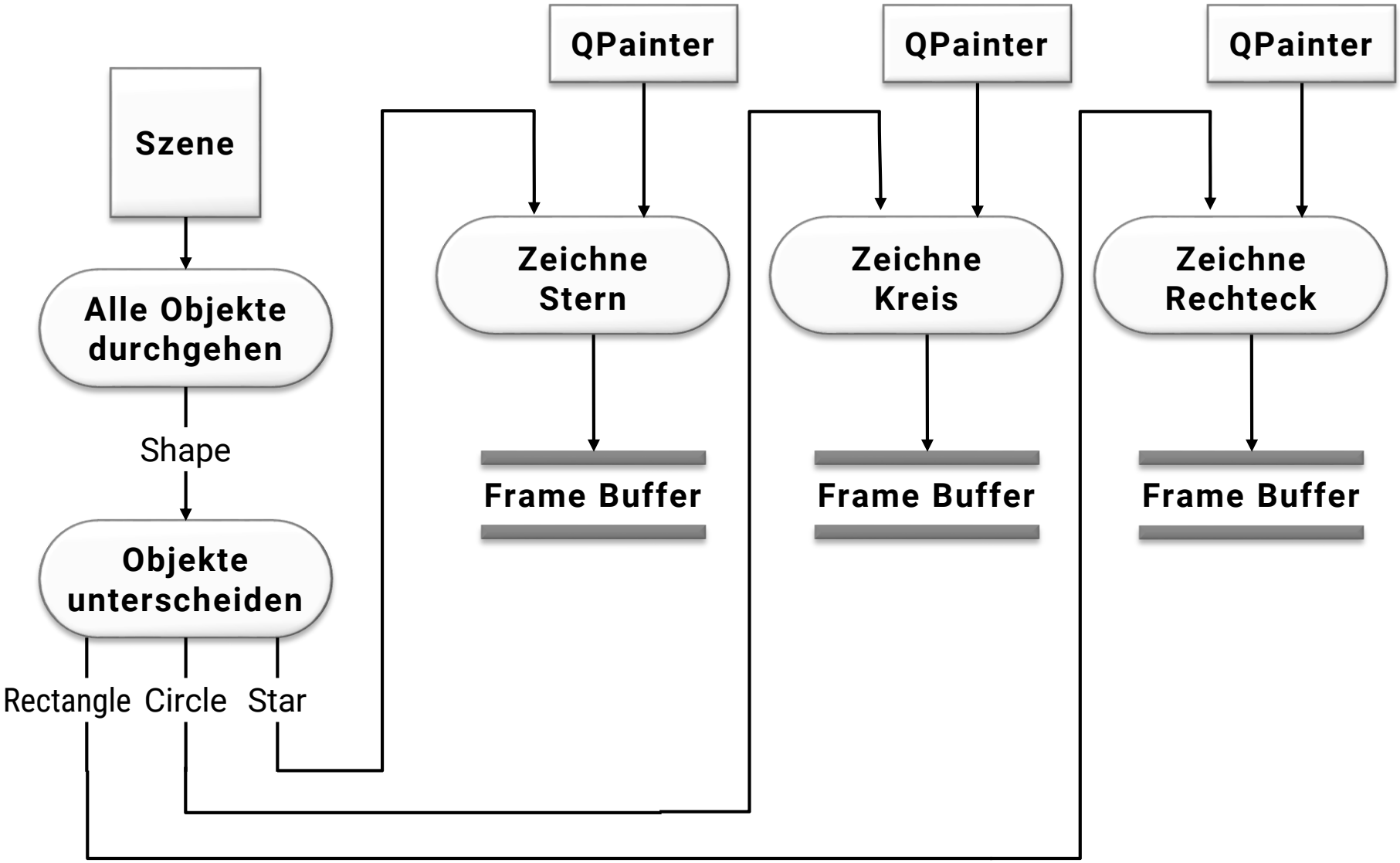
Welche Operationen braucht man?

Speziell in unserem Fall

- Geometrische Operationen
  - Geometrische Transformationen
    - Verschieben
    - Drehen, Skalieren (hier optional)
- Darstellung auf dem Bildschirm
  - Bei uns: Zeichnen mit Hilfe von `QPainter`

Entwurf Vektorgraphik-Bibliothek: Zeichenalgorithmus

Design: Zeichnen der Szene



Code für Zeichnen

```
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!

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)
```

Schwächen des prozeduralen Ansatzes

(für unsere Anwendung)

Unterprogramme

Wir brauchen

- Unterprogramme für
  - Erzeugen
  - Verändern (Geometrie auslesen und setzen)
  - Transformieren
  - Darstellen

  von allen Formen (Shape)
- Das ist eine Menge Arbeit
  - Die Unterprogramme sehen sich recht ähnlich
  - Immer Fallunterscheidung voran
  - Fehleranfällig (zumindest ohne exhaustivity-check)
  - Nicht einfach erweiterbar (Plug-Ins / Code nachladen unmöglich)

Nächster Schritt: OOP

Objektorientiertes Design

- Unterprogramme an Klassen binden
- Automatische Auswahl von Unterprogrammen
  - Polymorpher Methodenaufruf: „dynamic dispatch“
- Konstruktoren (& Destruktoren)
  - Automatisch sinnvolle Objekte bauen (Python __init__)

Ergebnis

- Mehr Ordnung im Code
- Weniger Fehleranfällig
- Einfacher um neue Typen erweiterbar (auch dynamisch)