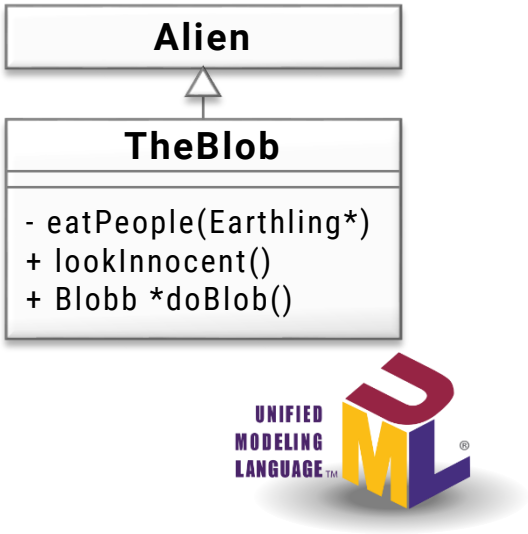
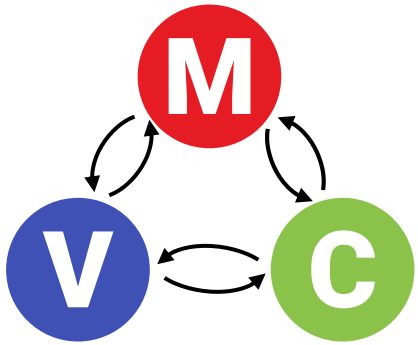
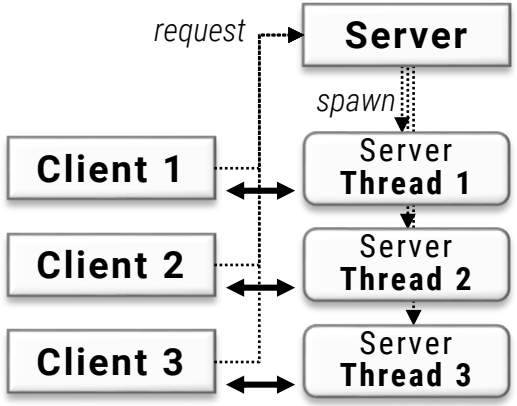
Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020





Design **Patterns**



Architectural **Patterns**

Foliensatz #12

Software Design & Muster: OOP & FP

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Designmuster für größere Systeme

Inhalt

- OOP vs. Functional – die Theorie
- Ein typisches “Standard” OOP Design
- Eine funktionale Variante davon (in OOP implementiert)
- Das “Expression Problem” und der eigentliche Trade-Off beim Entwurf

Der ganze Ärger startet mit…

Funktionsvariablen

Idee

Programme aus Teilen zusammensetzen

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

Bislang: Unterprogramme (C++: „Funktionen“)

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

Abstraktionen

Unterstützung durch Programmiersprache

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

Zwei Möglichkeiten

- Dynamische Übersetzung/Evaluation **(allgemeiner)**
  - Möglich in z.B. Python, LISP, C#
  - Code wird zur Laufzeit neu Erzeugt
- Funktionszeiger (statisch, **eingeschränkter**)
  - Verweise auf bestehende Unterprogramme als Variablen
  - Weniger flexibel, aber einfacher und weniger fehleranfällig

Dynamische Evaluation

Code zur Laufzeit erzeugen (Python)

```
# Variant 1a: Dynamic code evaluation
x = 21
# Output: 42
print( eval('x * 2') )

# Variant 1b: Dynamic code execution
# Output: Hello World
exec('print("Hello World")')

# Variant 1c: Code objects
code_obj = compile('print("Hello World")')
dis.dis(code_obj)
# Output: (see https://late.am/post/2012/03/26/exploring-python-code-objects.html)
#  0 LOAD_CONST   0 ('Hello, world')
#  3 PRINT_ITEM
#  4 PRINT_NEWLINE
#  5 LOAD_CONST   1 (None)
#  8 RETURN_VALUE
```

Funktionszeiger

Funktionszeiger / Funktionsobjekte (Python)

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

# Funktionsvariable
add_a_suitable_amount = add42

# Benutzung
print(add_a_suitable_amount(1337))

# Interessantere Anwendung
def apply_to_list(some_list, operation):
    for i in range(len(some_list)):
        some_list[i] = operation(some_list[i])
```

Funktionszeiger

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;
   // Alternativer Syntax
   // int(*add_a_suitable_amount)(int);
   // add_a_suitable_amount = &add42;
   // Seit C++11 auch… :-)
   // auto add_a_suitable_amount = &add42;

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

…in Unterprogrammen

```
int map(IntToIntFunc op,
        std::vector list) {
  for (int i=0; i<list.size(); i++) {
    list[i] = (*op)(list[i]);
  }
}
```

Funktionszeiger

Funktionszeiger / Funktionsobjekte (Object Pascal)

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

(* Object Pascal / Delphi *)
TYPE IntToIntFunc = FUNCTION(x : INTEGER) : INTEGER;
(* FYI: Modula 2 wäre ähnlich, kennt aber nur Prozeduren *)
TYPE IntToIntFunc = POINTER TO PROCEDURE(x: INTEGER, VAR result: INTEGER);

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

Zurück zum Konzeptionellen…

Funktionen

Funktion

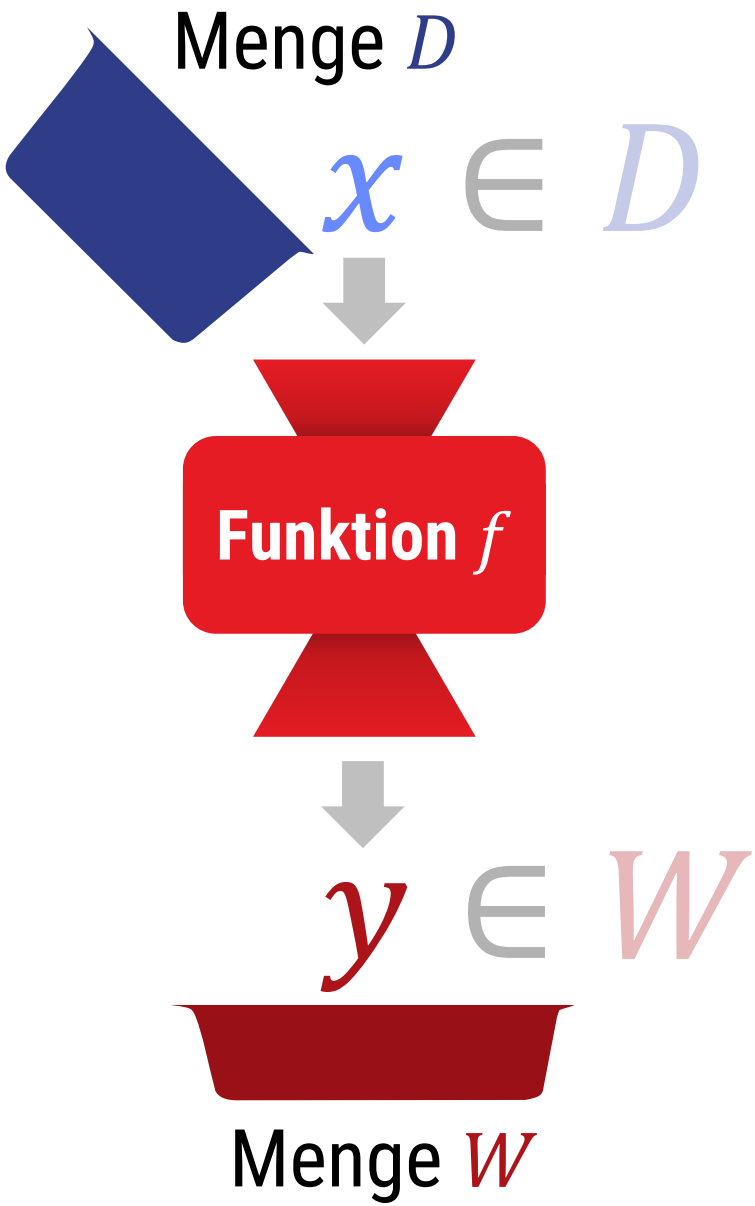
- Zuordnung von Elementen

$$f : D \to W$$

$$x \mapsto y = f(x)$$

$$x \in D, \quad y \in W$$

- Deterministisch
  - Gleiche Eingabe → gleiche Ausgabe
- Ergebnis muß für alle Eingaben definiert sein
  - D = “Definitionsmenge” (domain)
  - W = “Ziel/Wertemenge” (codomain / target set)



Funktionen

Zusammengesetzte Ein- / Ausgabe

- Signatur:

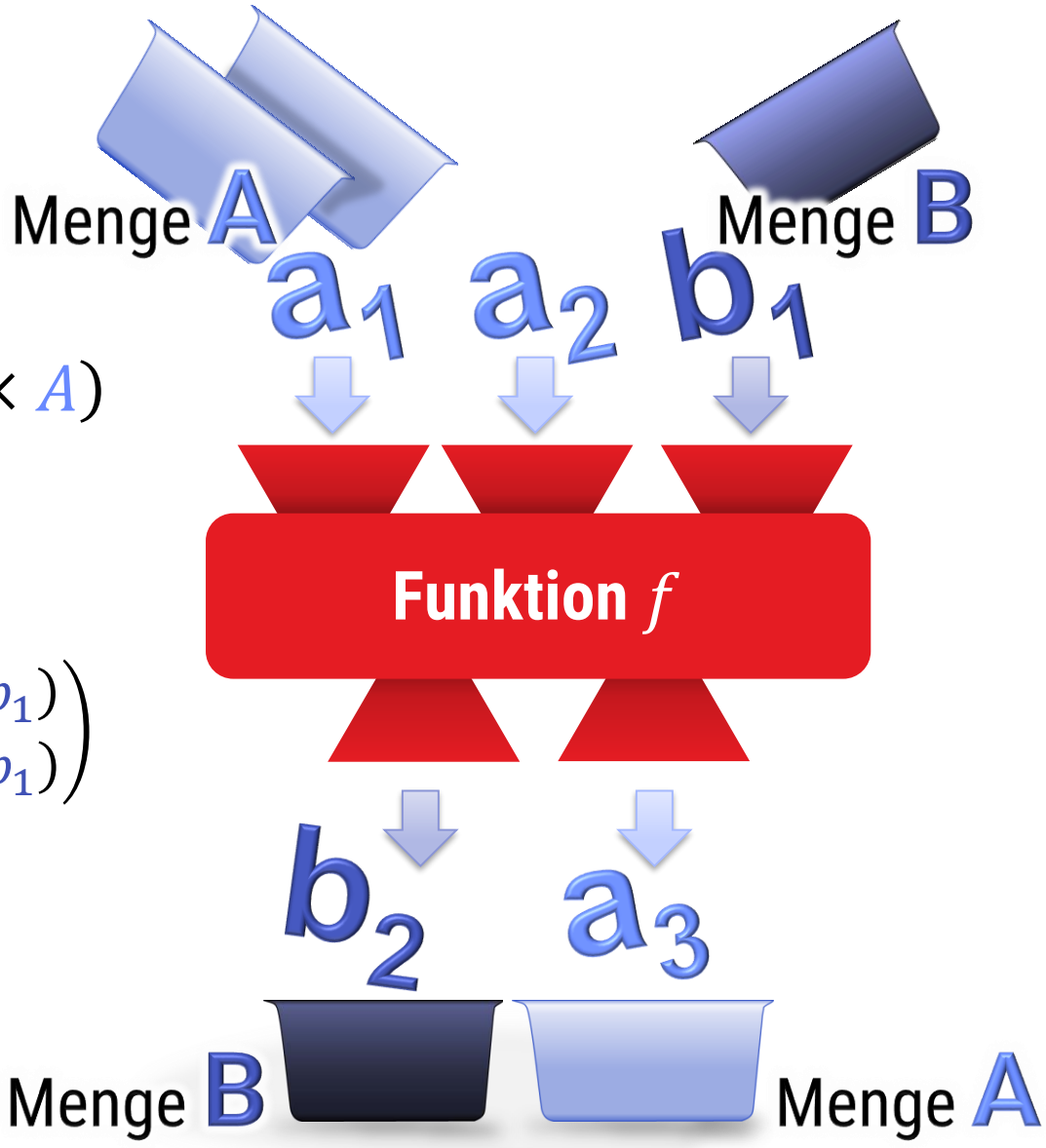
  $f : (A \times A \times B) \rightarrow (B \times A)$

- Zuordnung:

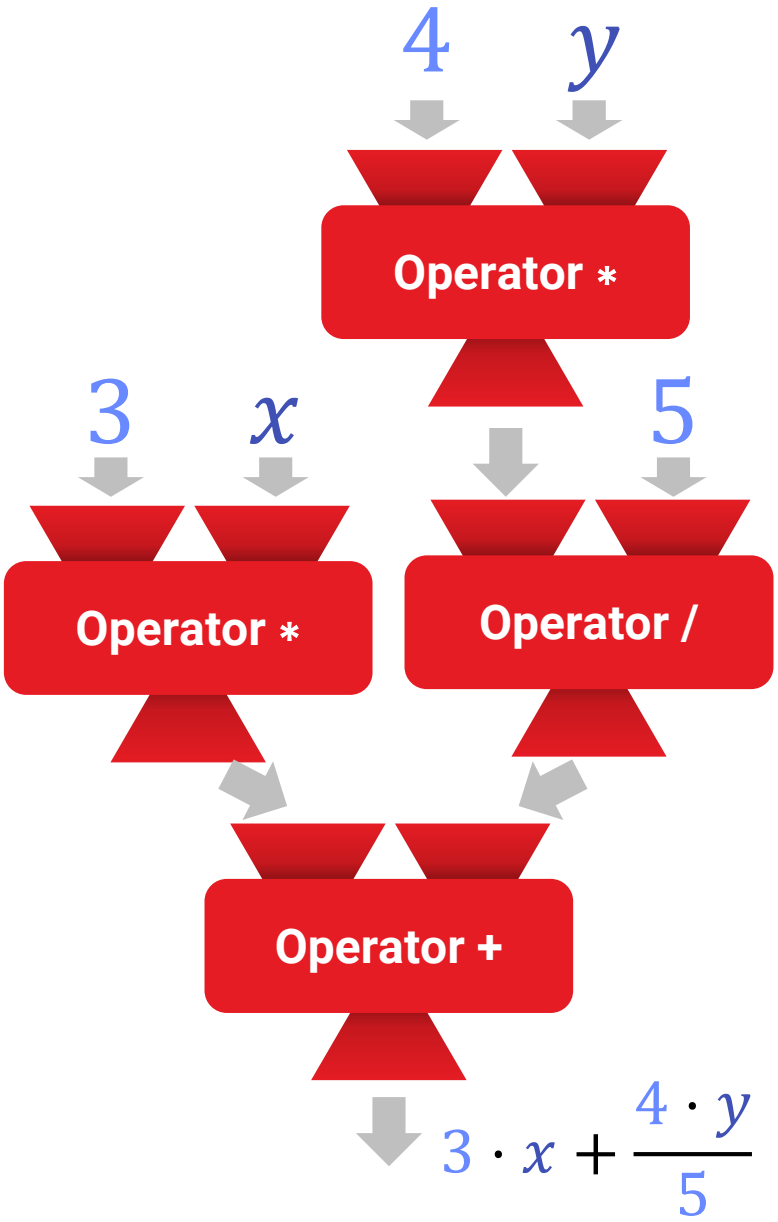
  $a_1, a_2, b_1 \mapsto f(a_1, a_2, b_1)$

  $= \begin{pmatrix} f_1(a_1, a_2, b_1) \\ f_2(a_1, a_2, b_1) \end{pmatrix}$

  $= \begin{pmatrix} b_2 \\ a_3 \end{pmatrix}$

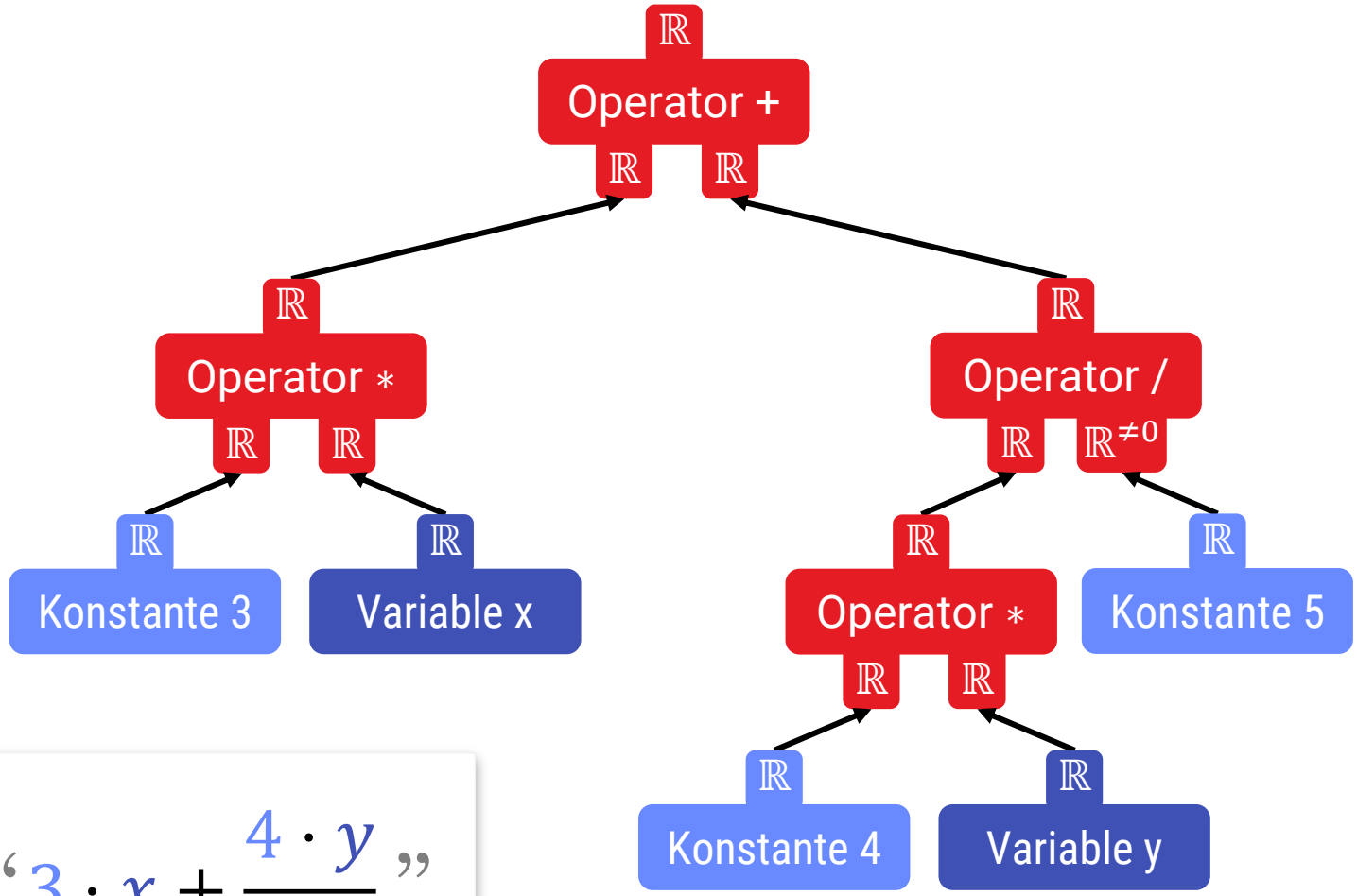


Arithmetische Ausdrücke



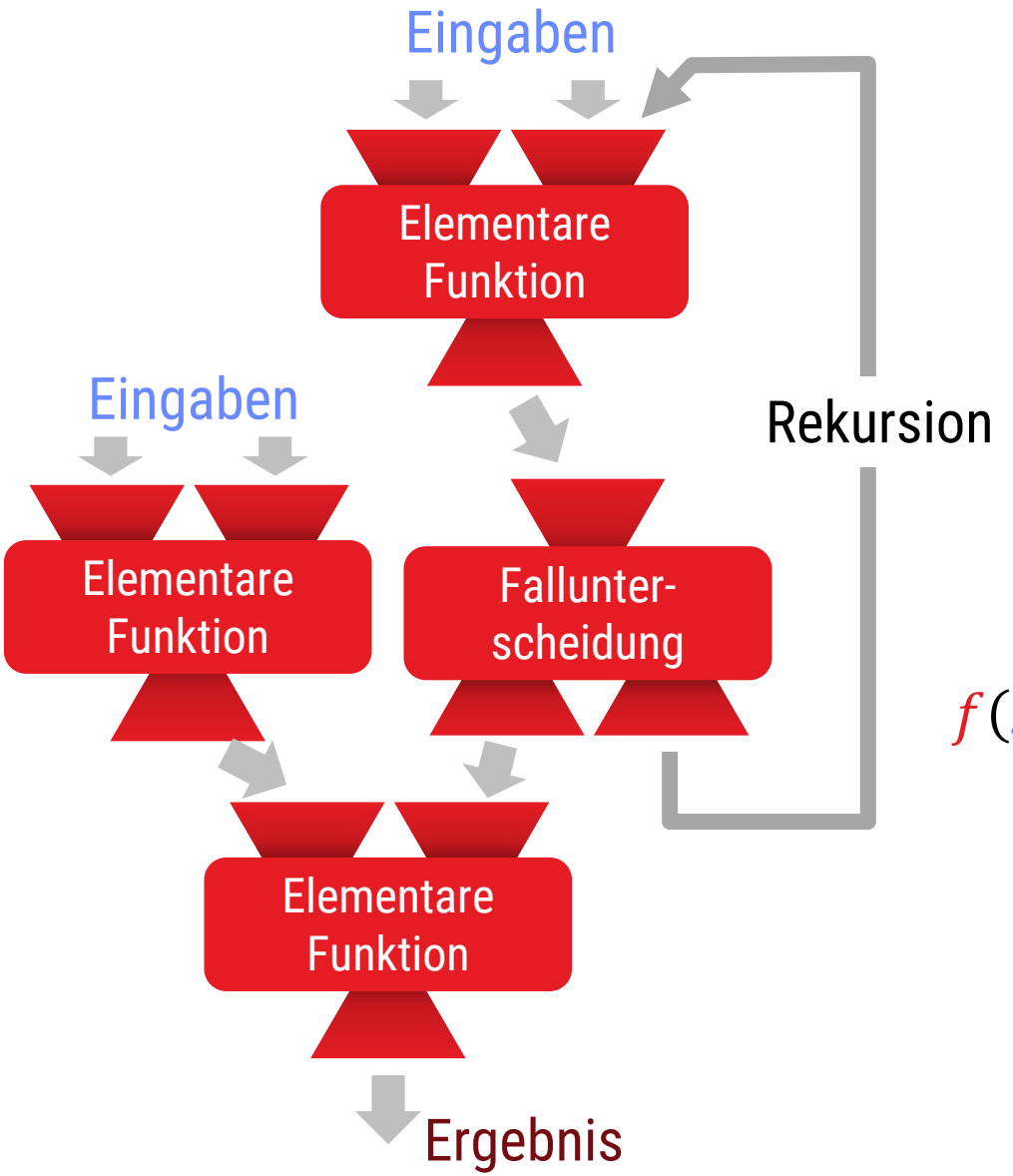
“$3 \cdot x + \frac{4 \cdot y}{5}$”

Statisch Typisiert



$$\text{“}3 \cdot x + \frac{4 \cdot y}{5}\text{”}$$

Mathematische Algorithmen



Bausteine

- Elementare Funktionen
- Rekursion

Beispiel

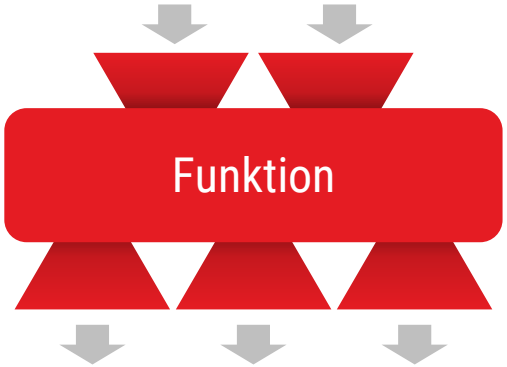
$$f(x) = \begin{cases} 1, \text{ falls } x = 0 \\ x \cdot f(x-1), \text{ sonst} \end{cases}$$

Informatik

- „Funktionale“ Programmierung

Funktionen als Bausteine

Bausteine



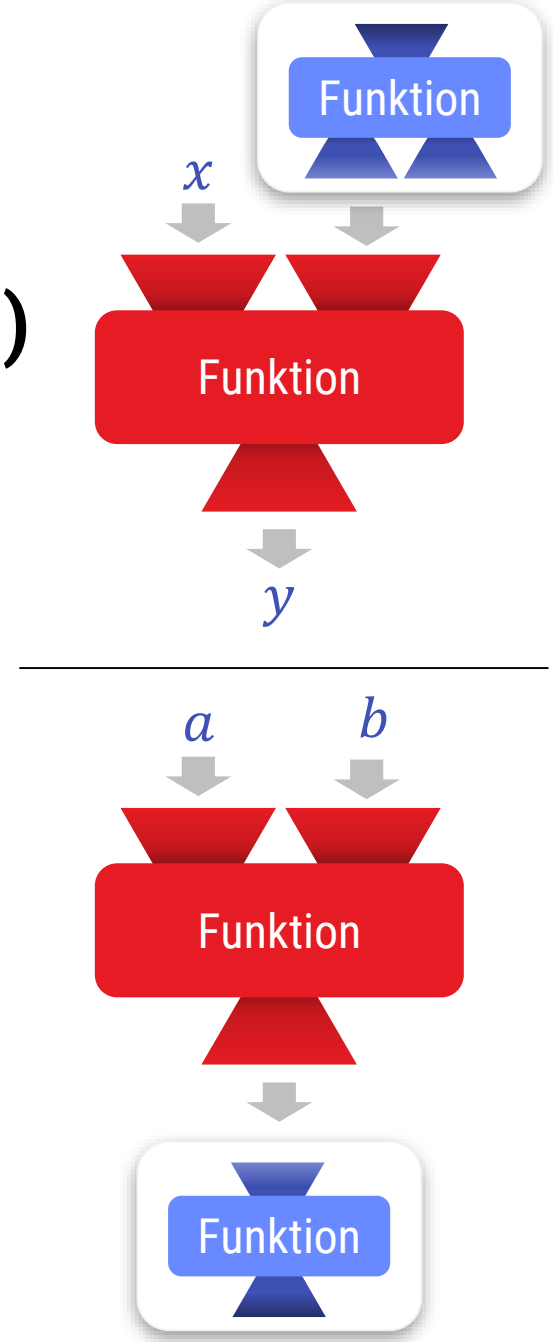
- Funkionen kapseln „Codeschnipsel“
- Definierte Schnittstelle
  - Ein- und Ausgabe spezifiziert
  - Statisch typisiert (in C, C++, Pascal, Modula 2, etc.)
  - Dynamisch typisiert in Python, JavaScript u.ä.
- Composition von Funktionen ergibt bei Bedarf ein „neu zusammengestelltes“ Programm

Higher Order…

Higher-Order Functions
(dt.: Funktionen höherer Ordnung)

- Funktionen, die andere Funktionen
  - Als Parameter übergeben bekommen
  - Als Rückgabe-„Wert“ haben
  - Oder beides

→ **Konfiguration zur Laufzeit**

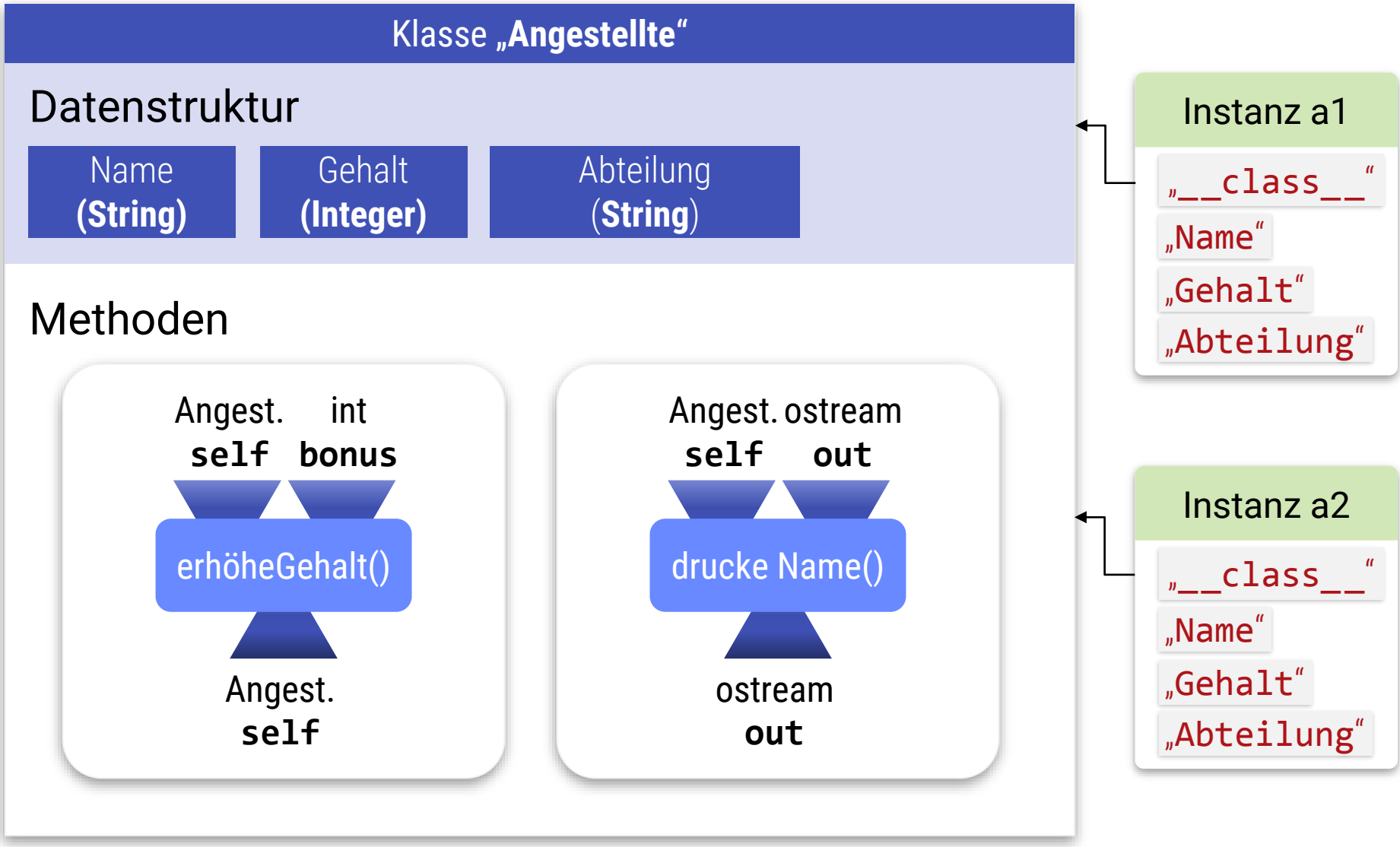


Zwei Schulen

(1) Objektorientierte Programmierung

- Abstrakte Datentypen
  - Zugriff auf Daten nur via Prozeduren/Funktionen/Methoden
  - Genaues Datenlayout „privat“
- Jeder Datentyp hat eine Tabelle geeigneter Funktionen
  - Aufruf via Namen
    - Direkt (Python) oder indirekt (C++/JAVA/Object Pascal)
  - Datentype ist für Objekte bekannt
    - Auswahl des richtigen Codes
  - „Vererbung“ erleichtert Implementation und Typprüfung

OOP

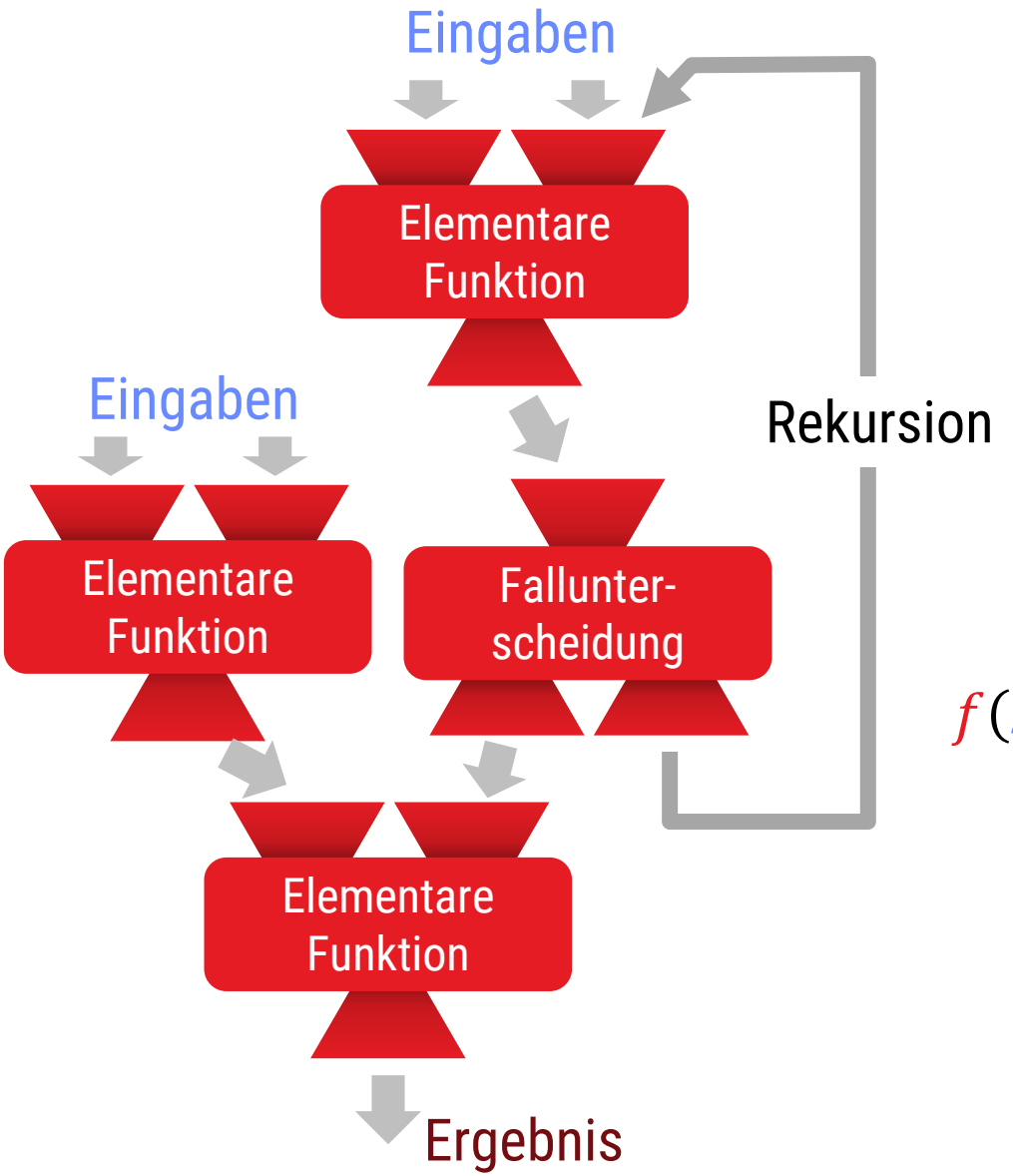

Klasse „Angestellte“
Datenstruktur
Name (String)
Gehalt (Integer)
Abteilung (String)
Methoden
Angest. self
int bonus
erhöheGehalt()
Angest. self
Angest. ostream self out
drucke Name()
ostream out
Instanz a1
„__class__“
„Name“
„Gehalt“
„Abteilung“
Instanz a2
„__class__“
„Name“
„Gehalt“
„Abteilung“

Zwei Schulen

(2) Funktionale Programmierung

- Aggressiver Einsatz von Funktionszeigern
  - Nicht nur an Typ von Daten gekoppelt
  - Code so parametrisierbar wie möglich
    - Higher-order functions
- Naiv führt dies zu Problemen
  - Seiteneffekte, unübersichtlich, schwer zu handhaben
- Typische Lösung
  - Einschränkung der Programmkonstrukte
    - Vermeiden von Zuweisungen
    - Rekursion statt Schleifen
- Abstrakter & schwerer zu lernen, aber sehr mächtig

Funktionale Programmierung



Bausteine

- Elementare Funktionen
- Rekursion

Beispiel

$$f(x) = \begin{cases} 1, \text{ falls } x = 0 \\ x \cdot f(x-1), \text{ sonst} \end{cases}$$

OOP vs. Functional

Was tun im Jahre 2018?

- Im Internet tobt teilweise der heilige Krieg
- Der Profi kennt & beherrscht alles :-)

OOP (Fokus in EiS laut Modulhandbuch)

- Für viele „Standardprobleme“ sehr gut geeignet
- z.B. Komponentenarchitekturen, GUIs

Funktional (Vorlesung „Programmiersprachen“)

- Die Idee ist sehr mächtig
- Kenntnis verbessert auch OOP-Programmierstil

Standard OOP Design

Beispiele

Drei Beispiele

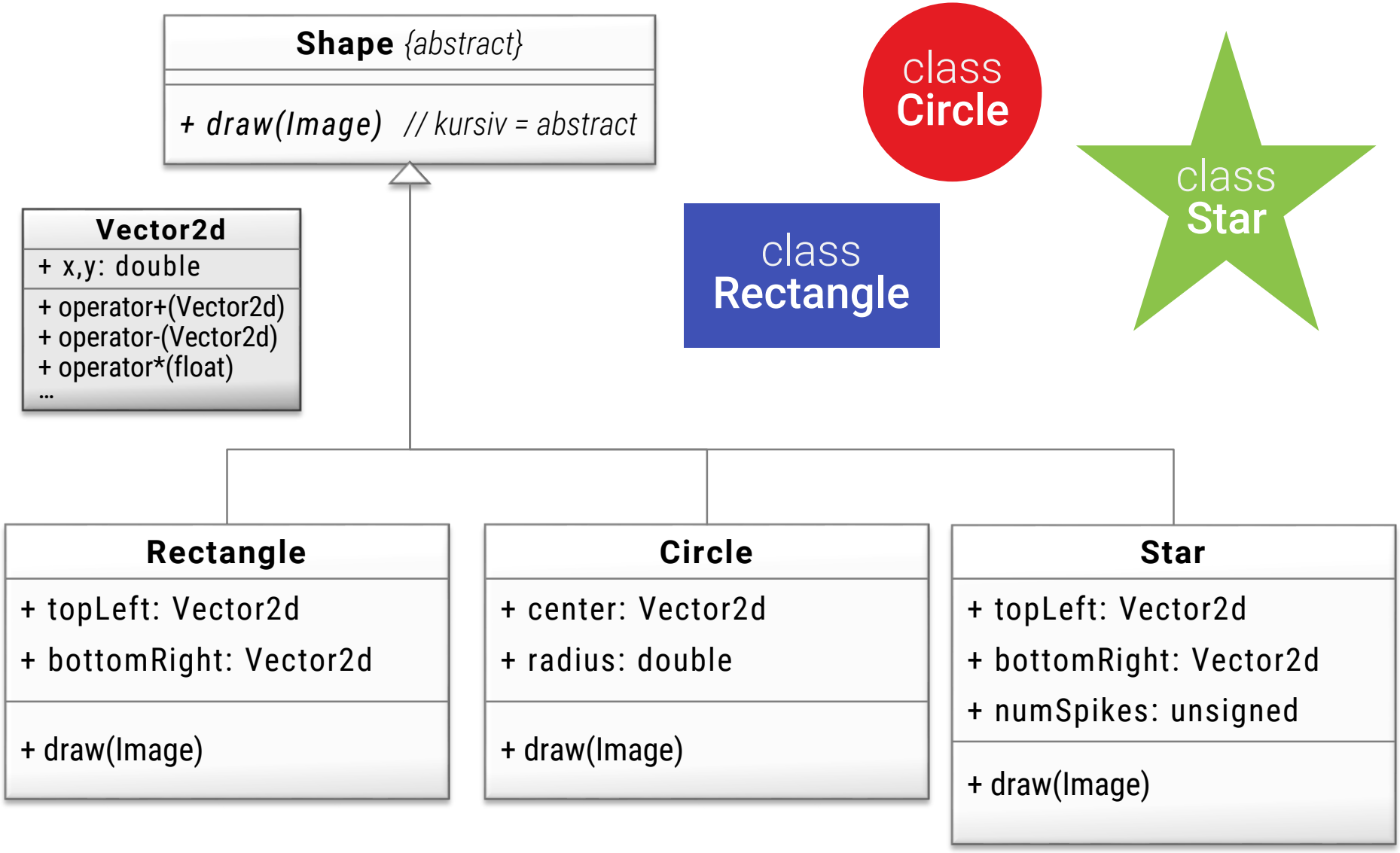
- Standard-OO Architektur: **Typisierte hierarchische Komposition mit Subtyping**
  - Komplizierter Name – einfache Idee
  - Beispiel: Vektorzeichenprogram
- Flexiblere Variante: **Datenflussgraphen**
  - „Funktionale“ Idee
  - Komplexer aber flexibler
  - Beispiel: Wieder ein Vektorzeichenprogramm
- Ereignisse dazu: **GUI Frameworks**
  - Event-Loops, hierarchische Propagation und Delegation
  - Asynchrone Programmierung

Beispiel

Beispiel:

- Objektorientierte Implementation eines „Vektororientierten Zeichenprograms“
  - Objekte repräsentierten geometrische Formen
  - „Vektorgraphik“ – die Objekte können sich selbst in jeder Auflösung darstellen
  - Bitmaps nur als Ausgabe

Grobe Architektur für komplette Anwendung


Shape {abstract}
+ draw(Image) // kursiv = abstract
class Circle
class Star
Vector2d
+ x,y: double
+ operator+(Vector2d)
+ operator-(Vector2d)
+ operator*(float)
…
class Rectangle
Rectangle
+ topLeft: Vector2d
+ bottomRight: Vector2d
+ draw(Image)
Circle
+ center: Vector2d
+ radius: double
+ draw(Image)
Star
+ topLeft: Vector2d
+ bottomRight: Vector2d
+ numSpikes: unsigned
+ draw(Image)

Architektur

Shapes

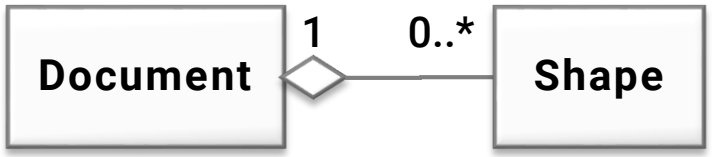
- Zeichnen sich selbst (erzeugen Pixelbild)
- Eigener Zeichenalgorithmus für jedes „Shape“

Nachfahren

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

Zeichenalgorithmus

- Dokument = Liste von „Shapes“
- For each: `draw()`



Beispiel-Code (grob)

```
class Document {
 private:
   std::vector<Shape*> shapes;
 public:
   // draws all shapes
   virtual void draw(Graphics *g) {
      for (int i=0; i<shapes.size(); i++) {
         shapes[i]->draw(g);
      }
   }
};
```

Beispiel-Code (grob)

```
class Document {
 private:
   std::vector<Shape*> shapes;
 public:
   // draws all shapes
   virtual void draw(Graphics *g) {
      for (Shape* s: shapes) { // Iterator-Syntax für std::vector (C++11)
         s->draw(g);           // Analog zu Python "for s in shapes:"
      }
   }
};
```

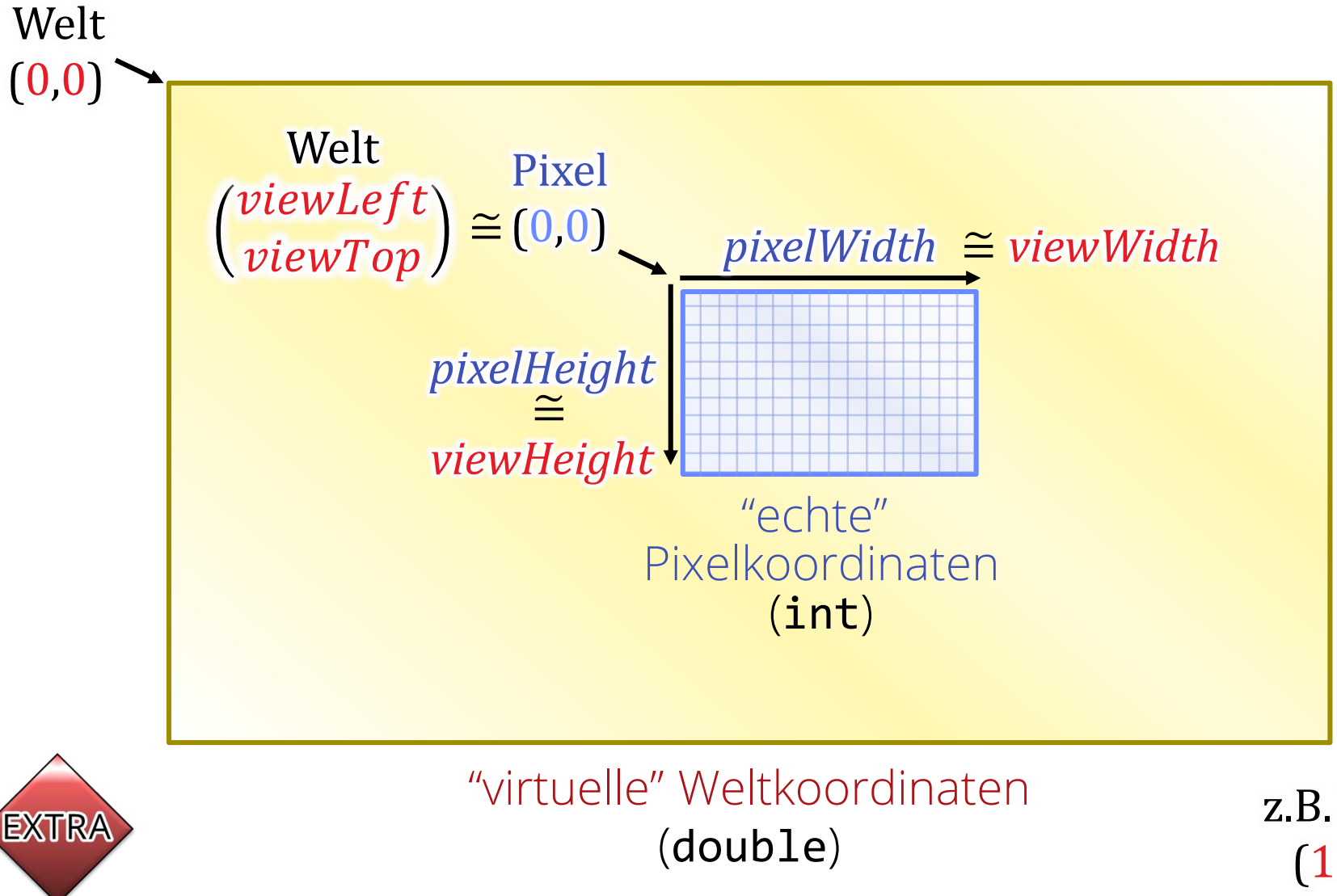
„Graphics“

```
class Graphics {
 private:
   // Bitmap Framebuffer
   struct Vector3ub {uint8_t r,g,b}; // Pixel Type
   std::vector<Vector3ub> pixels;    // Framebuffer (linearized)
   unsigned pixelWidth, pixelHeight; // width and height of framebuffer

   // Virtual viewport (floating point, "vector graphics")
   double viewTop,viewLeft;          // top left corner of virtual viewport
   double viewWidth,viewHeight;      // width and height of virtual viewport

 public:
   // Example operation
   void drawPixel(double view_x, double view_y, Vector3ub color) {

      // Projection / window transformation
      int pixel_x = (view_x - viewLeft) / viewWidth * pixelWidth;
      int pixel_y = (view_y - viewTop) / viewHeight * pixelHeight;

      // Clipping
      if (pixel_x>=0 && pixel_x<width && pixel_y>=0 && pixel_y<height)
         // Drawing (setting the pixel)
         pixels[pixel_y*pixelWidth + pixel_x] = color;

      }
   }

   void drawLine(Vector2d start, Vector2d end, Vector3ub color) {…}
   void drawCircle(Vector2d center, double radius, Vector3ub color) {…}
   void drawText(Vector2d start, std::string text, Vector3ub color) {…}
   ...
};
```

Weitere Abstraktion

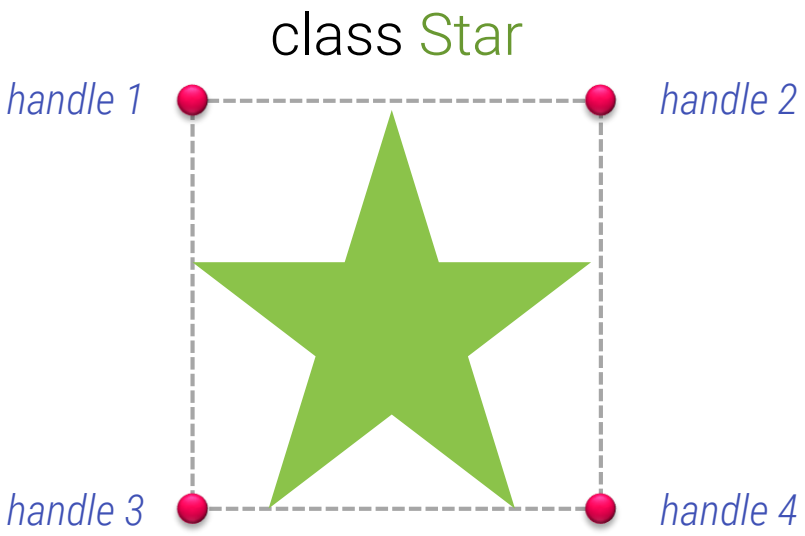
```
/// Abstract base class
class Graphics {
 public:
   virtual unsigned getPixelWidth() = 0;
   virtual unsigned getPixelHeight() = 0;
   virtual double viewTop() = 0;
   virtual double viewLeft() = 0;
   virtual double viewWidth() = 0;
   virtual double viewHeight() = 0;
   virtual void drawPixel(double view_x, double view_y, Vector3ub color) = 0;
   void drawLine(Vector2d start, Vector2d end, Vector3ub color) = 0;
   void drawCircle(Vector2d center, double radius, Vector3ub color) = 0;
   void drawText(Vector2d start, std::string text, Vector3ub color) = 0;
   ...
};

/// In-memory (offscreen) bitmap image
class Bitmap : public Graphics {
 public:
   ...
   virtual void drawPixel(double view_x, double view_y, Vector3ub color) {…}
   ...
};

/// Part of a window on screen
class WindowArea : public Graphics {…}

/// Maybe a Laser printer (converting to Postscript)
class Printer : public Graphics {…}
```

„Viewport Transformation“



EXTRA

Vertiefung



Shape *{abstract}*

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

Rectangle

+ topLeft: Vector2d
+ bottomRight: Vector2d

+ draw(Image)
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d

Circle

+ center: Vector2d
+ radius: double

+ draw(Image)
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d

Star

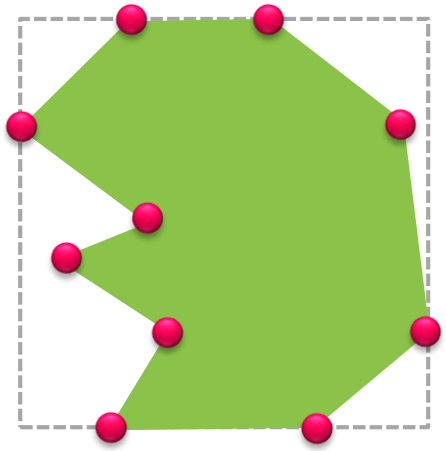
+ topLeft: Vector2d
+ bottomRight: Vector2d
+ numSpikes: unsigned

+ draw(Image)
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d

Abstrakte Interaktion

Selektion von Objekten

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

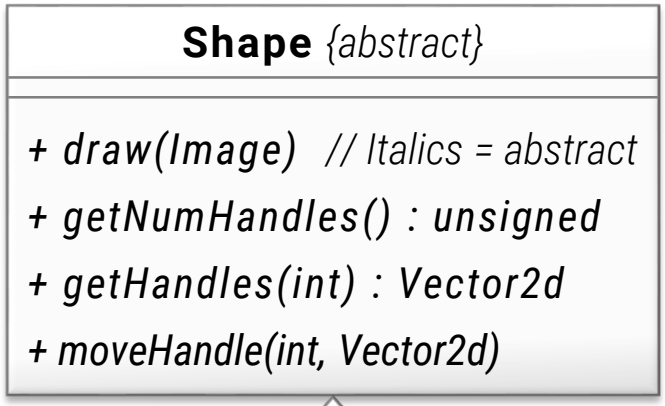


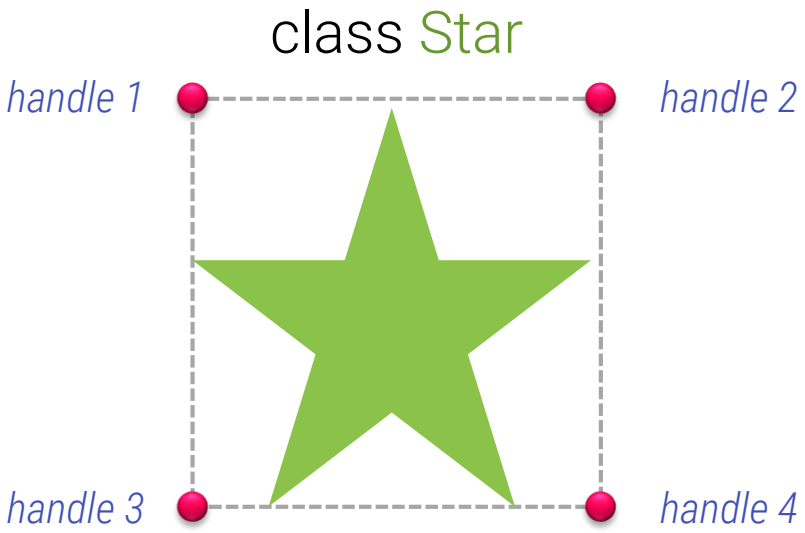
Polymorpher Algorithmus

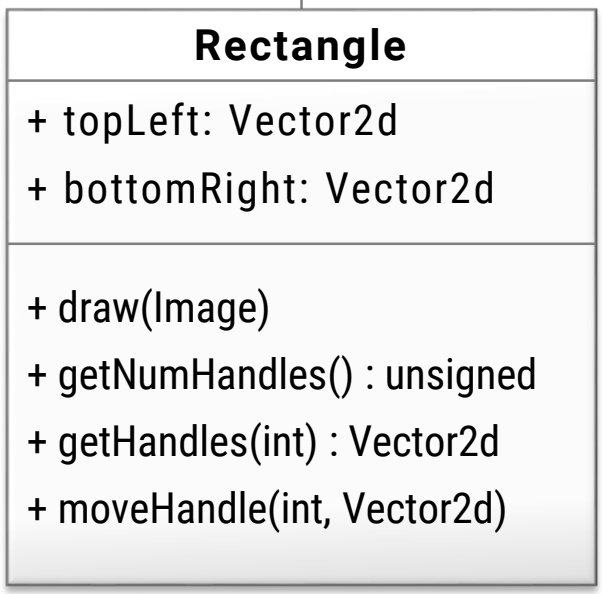
- Für alle Objekte in Dokument
- Teste, ob angeklickt
  - Wenn ja, markiere als selektiert
- Für selektierte Objekte werden Handles gezeichnet

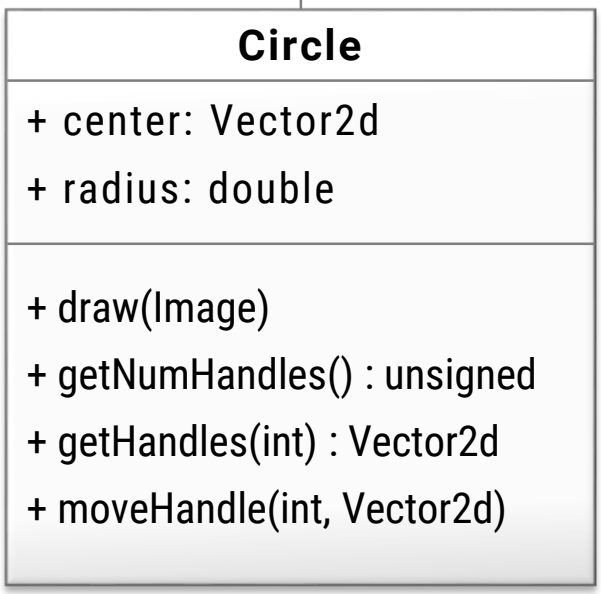
Beispiel-Code (grob)

```
// find closest shape by handle (“picking”)
void pickClosestHandle(Graphics *view, Document *doc, // scene
                       int mouse_x, int mouse_y,      // mouse coordinates
                       int &shape, int &handle, bool &found) // result
{  // convert pixel (mouse) coordinates to view (world) coordinates
   double m_x_view = mouse_x * view->getViewWidth() / view->getPixelWidth()  + view->getViewLeft();
   double m_y_view = mouse_y * view->getViewHeight() / view->getPixelHeight() + view->getViewTop();
   Vector2d mouseView = Vector2d (m_x_view, m_y_view);
   float minDist = 1E20f; // should be far enough…
   for (int i=0; i<doc->getNumShapes(); i++) {
      Shape *shape = document->getShape(i);
      for (int j=0; j<shape->getNumHandles(); j++) {
         Vector2d handlePoint = shape->getHandle(j);
         // Distance = sqrt((x1 - x2)^2 + (y1 - y2)^2) (Pythagoras)
         float dist = Vector2d::distance(handlePoint, mouseView);
         if (dist < minDist) {
            found = true; shape = i; handle = j; minDist = dist;
         }
      }
   }
}
```


Shape *{abstract}*
+ draw(Image) // Italics = abstract
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)


class Star
handle 1
handle 2
handle 3
handle 4


Rectangle
+ topLeft: Vector2d
+ bottomRight: Vector2d
+ draw(Image)
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)


Circle
+ center: Vector2d
+ radius: double
+ draw(Image)
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)


Star
+ topLeft: Vector2d
+ bottomRight: Vector2d
+ numSpikes: unsigned
+ draw(Image)
+ getNumHandles() : unsigned
+ getHandles(int) : Vector2d
+ moveHandle(int, Vector2d)

Abstrakte Interaktion

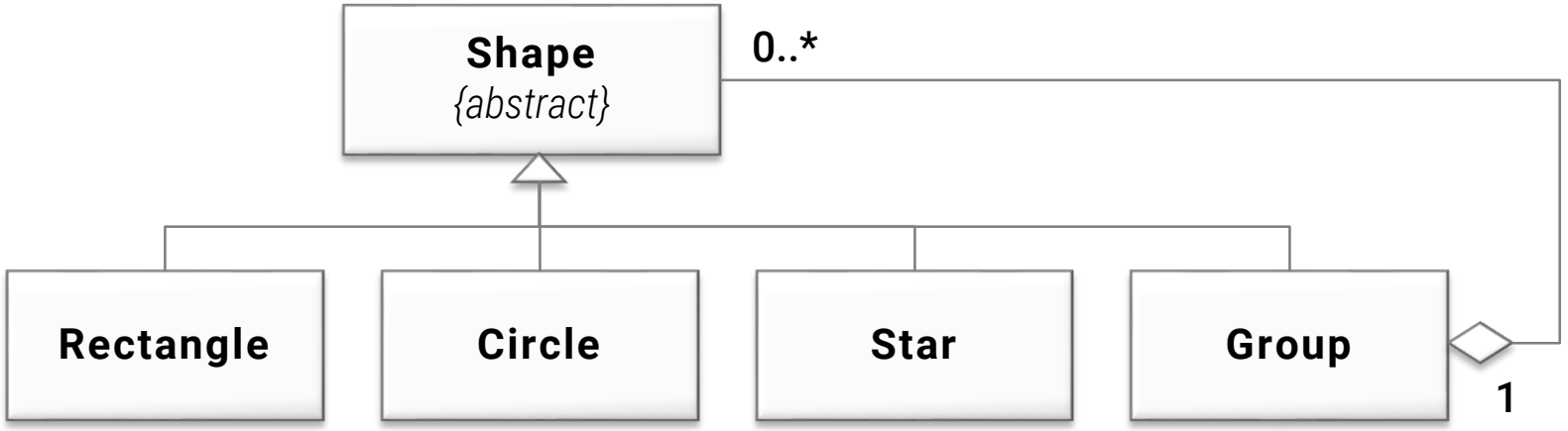
Handles werden mit der Maus „verschoben“

- 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

Listen von Objekten (Gruppierung)



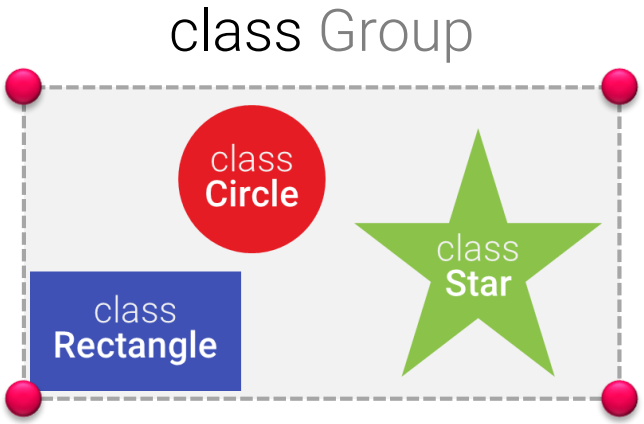
Indirekte Rekursion

Zeichenmethode

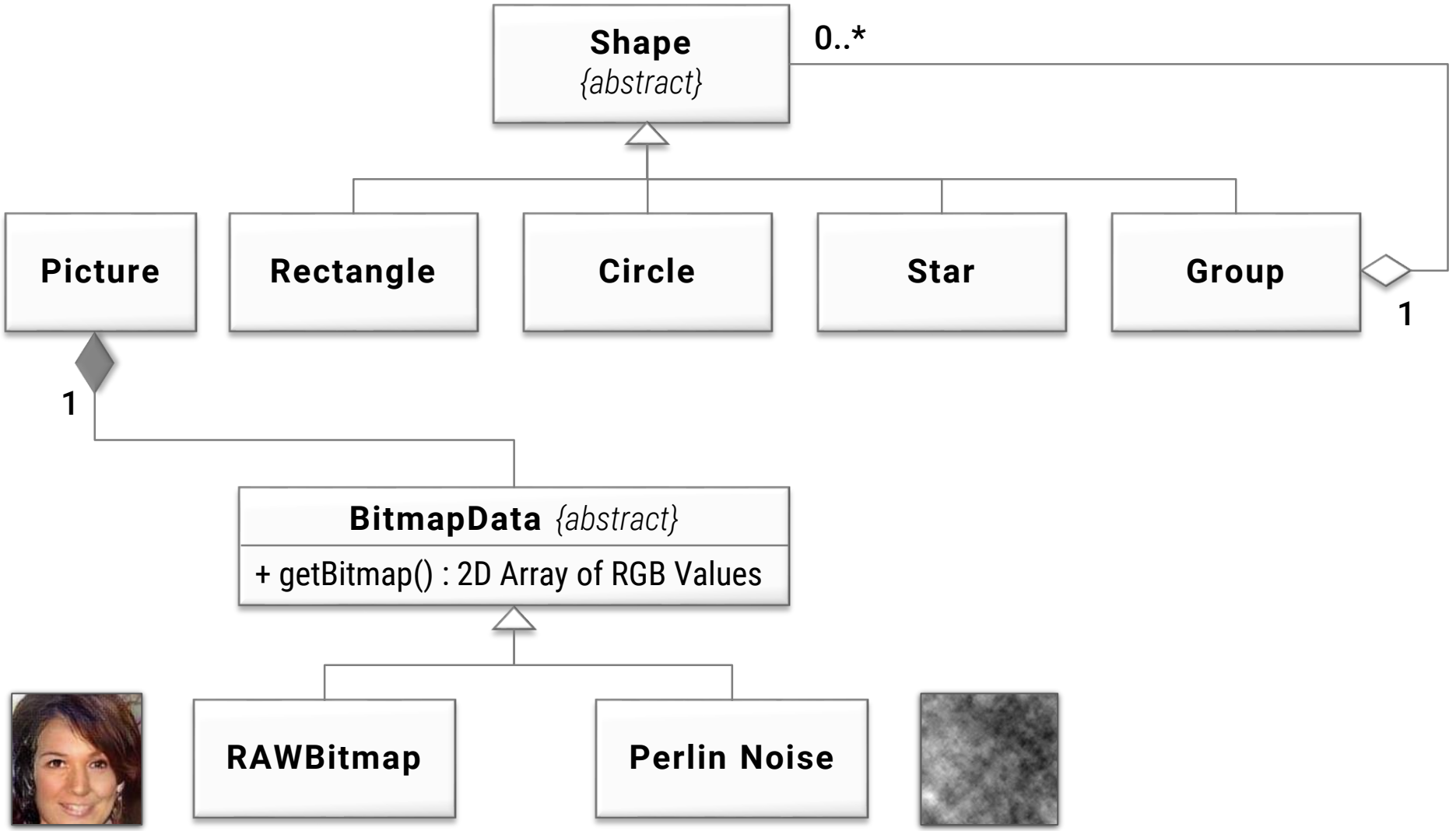
- Draw für „Group“ zeichnet alle enthaltenen Objekte
- Dies können auch Gruppen sein
- (Direkte und indirekte) Rekursion möglich

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

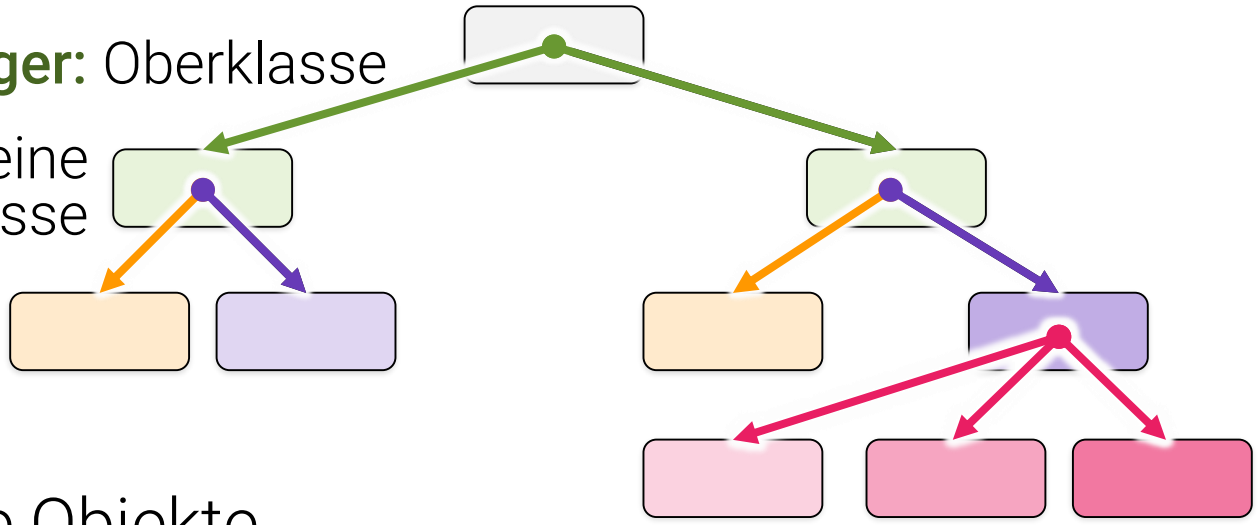
Was ist hier neu?

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)

Standard OOP Design:
Objekthierarchien, Serialisierung & Infrastruktur

Betrachte Zeichenprogramm



Baumstruktur

- Geschachtelte Objekte
- Im Beispiel: „Group“ als einziger innerer Knoten

Allgemein

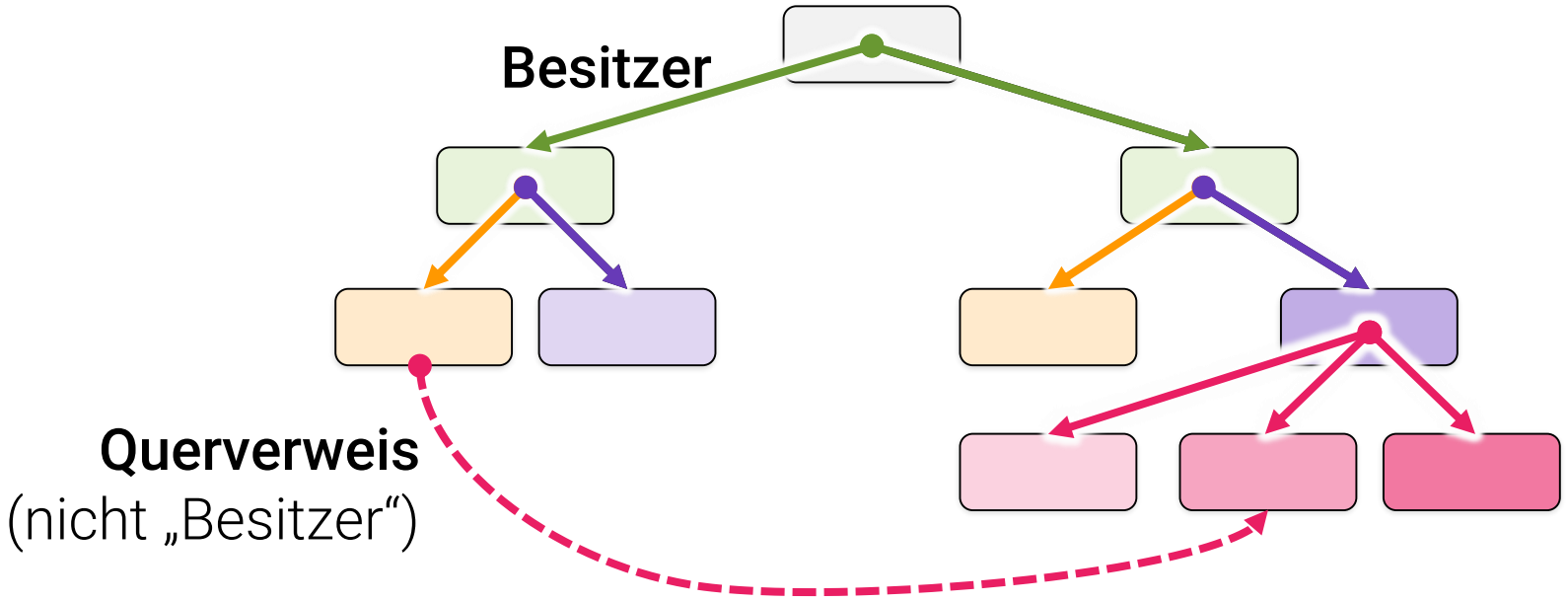
- Ähnlich wie Schachtelung in Programmiersprachen
  - Objekt-Membervariablen: Platzhalter für Ergänzungen
  - Oberklassen: Einschränkungen möglicher Typen

In C++ – Speichermanagement

In C++ manuelle Speicherfreigabe

- Baumstruktur als „Owner“-Struktur
  - Kindobjekte „gehören“ dem „Parent“-Objekt
  - Parents löschen Kindobjekte rekursiv im Destruktor
- Graphen von Objekten
  - Oft ein „Hauptbaum“ von Besitzern
  - Weitere Verweise sind keine „Owner“

„Parents are Owners“-Ansatz



Konsistenz bei Querverweisen

- Baumstruktur ist automatisch konsistent
- Bei Querverweisen „dangeling references“ möglich
  - Verweise auf bereits gelöschte Objekte

„Parents are Owners“-Ansatz

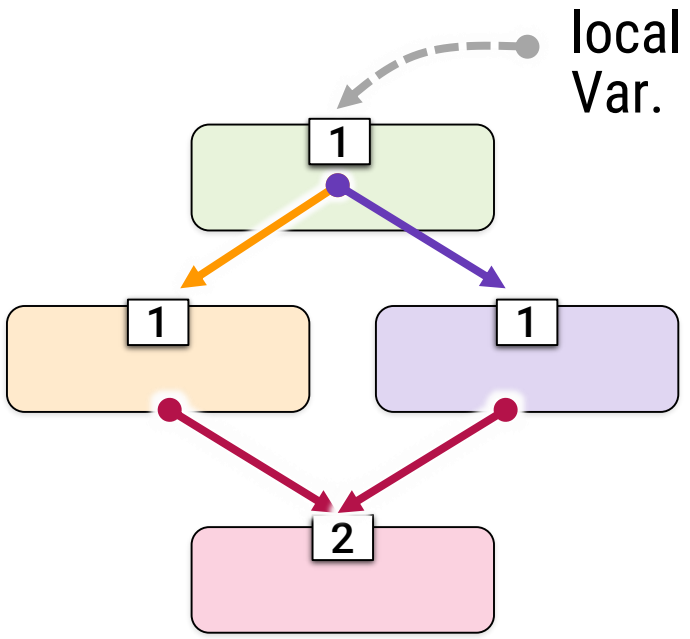
Lösungsvorschlag 1: Buchführen

- Bei Löschen von Objekten Verweise prüfen
  - Hilfsdatenstrukturen (z.B. Rückwärtszeiger) nötig, um Suche zu beschleunigen
- **Vorteil:** Schnell
- **Nachteil:** Kompliziert, etwas Overhead (Buchführung)

„Parents are Owners“-Ansatz

Lösungsvorschlag 2: Reference-Counting

- Mehrere Parents für das selbe Objekt möglich
- Alle potentielle Besitzer
- Zähler im Objekt zählt Anzahl der Verweise
- Objekt löscht sich selbst, wenn Zähler auf `0` geht
- **Vorteil:**
  schnell, recht allgemein
- **Nachteil:**
  Nur azyklische Graphen



„Parents are Owners“-Ansatz

Lösungsvorschlag 3: Strings

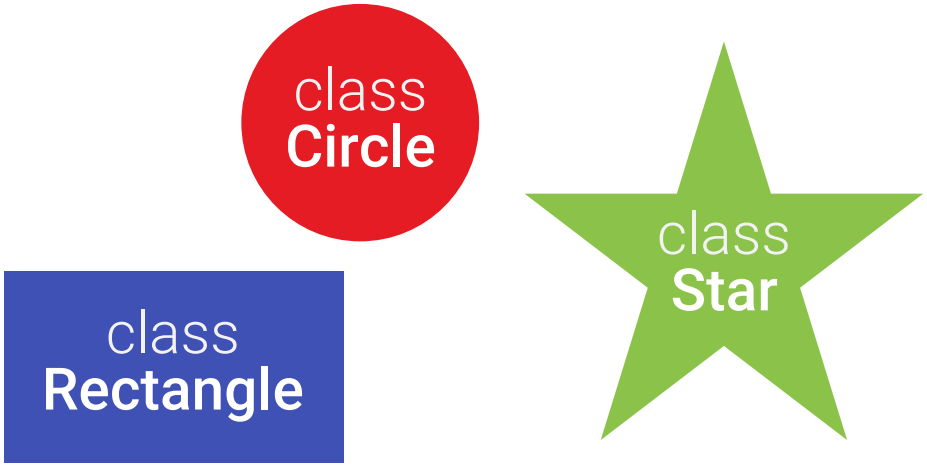
- Jedem Objekt lokal (in Bezug auf Parent) eindeutigen Namen geben
- Zugriff über Zeichenketten
  - Fehlerbehandlung, falls Name nicht gefunden
- Beispiele für symbolische Verweise
  - „`/Dokument-A/Paragraph[1]/Zeichen[7]/Format`“
  - „`Dokument-A.Paragraph[1].Zeichen[7].Format`“
- **Vorteil:** Sehr flexibel (auch als UI geeignet)
- **Nachteil:** Seeeeehhhhr langsam, Inkonsistenz möglich (kein Absturz, aber Fehler)

Serialisierung & OOP-Infrastruktur

Unvollständige Anwendung

Was fehlt unserem *Vektorzeichenprogramm* noch?

- Laden & Speichern von Dokumenten
- Kopieren von Objekten
- Besseres GUI



Evolution des Designs

Speichern Prozedural

- Funktionen
  „`void saveDocument(Document*, File*)`“
  „`Document *loadDocument(File*)`“
- Dateiformat definieren, z.B.

```
circle: radius=5.0, center = {2,3}
rectangle: topLeft = {1,0} topRight = {4,5}
…
```

- Fallunterscheidung für jeden Typ Shape, in etwa:

```
if (dynamic_cast<Circle*>(shape)) {…}
elseif (dynamic_cast<Star*>(shape)) {…}
…
```

Probleme Prozdural

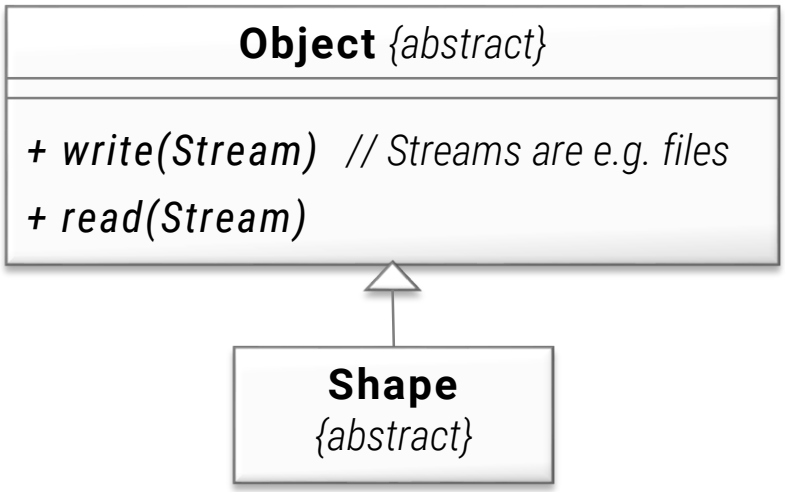
Diverse Probleme

- Schlecht erweiterbar
- Neue Shapes:
  - Änderung des Dateiformates
  - Fallunterscheidungen müssen eingefügt werden
  - Nicht vergessen!
- Nach Übersetzung nicht mehr möglich
  - Keine Plug-Ins möglich!
- Dateiformat handdefiniert
  - Inkonsistenzen möglich
  - Korr. Rekonstruktion nach save→load im Fehlerfall nicht garantiert

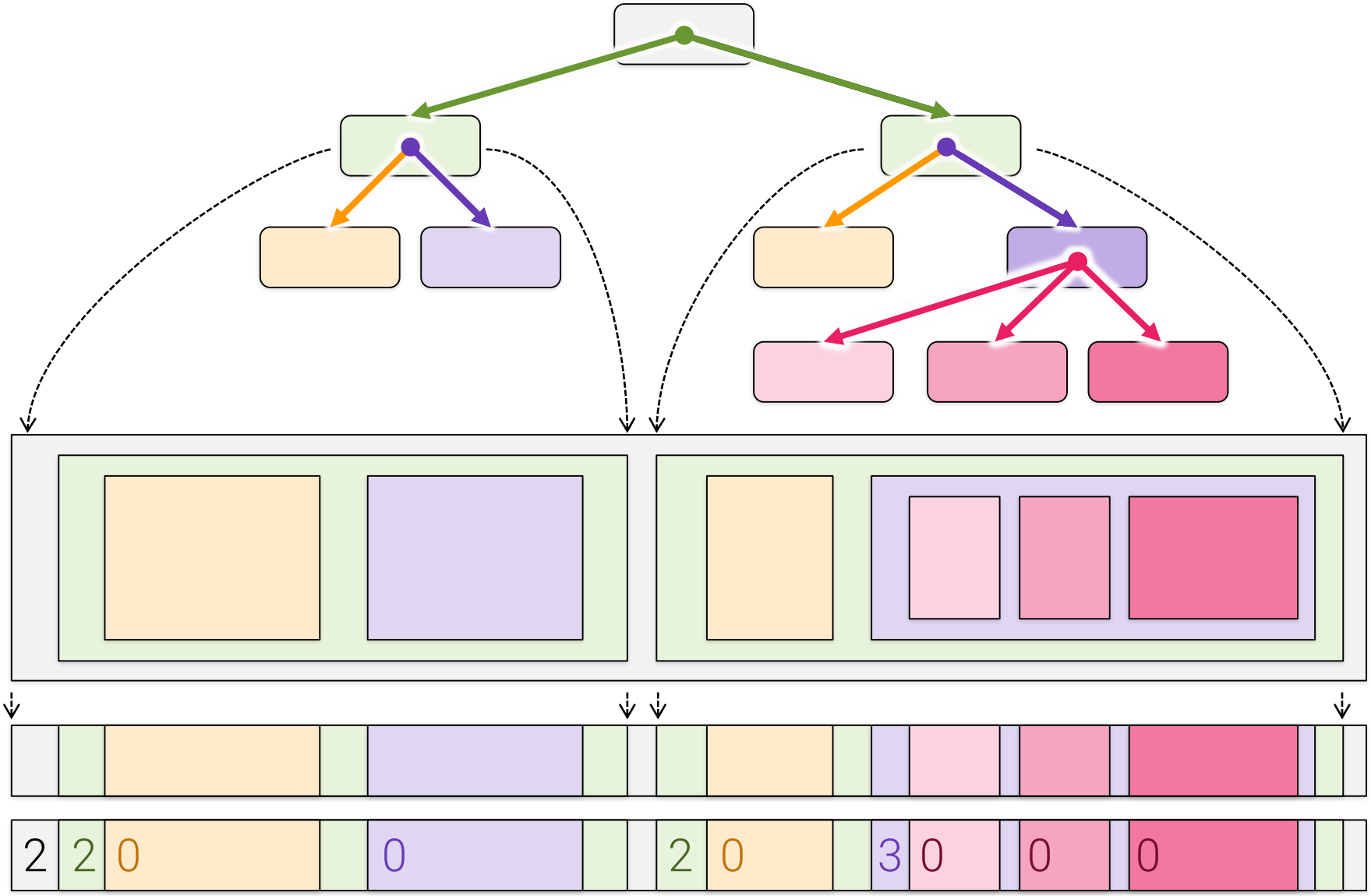
Evolution des Designs

Objektorientiert

- Member-Funktionen
  „`void read(Stream*)`“
  „`void write(Stream*)`“
  in Basisklasse!
- Methoden handgeschrieben
  - Schreibt alle Felder in Datei
  - Inkrementell erweitert in Nachfahrenklassen
- Konsistenz?
  - Jede Klasse garantiert, dass sie alle „Ihre“ Felder liest bzw. schreibt
  - Zusätzliche „Größenmarker“ um Fehler zu erkennen



„Serialisierung“



Beispiel: Objektorientiert

Schreiben eines „Shapes“

- Schreiben des eigenen Typs (z.B. Klassenname)
- Felder wie `radius`, `center`, etc.

Schreiben einer Liste wie „Group“

- Schreiben des eigenen Typs (z.B. als „`Group`“)
- Schreiben der Größe (als **`int`**)
- Danach Aufruf von `write()` für alle Unterobjekte

Beispiel

Datei (Stream)

```
Header
Class: "Group"
Number Items: 2
    Class: "Star"
    topLeft:
        Type: "Vector2d"
        x: 3
        y: 4
    bottomRight:
        Type: "Vector2d"
        x: 5
        y: 6
    numSpikes: 7
    Class: “Rectangle"
    topLeft:…
```

Format

- Grau: Erwartete Typen
- Blau: Werte/Zustand

Bemerkung

- Graue Daten können komplett ausgelassen werden
- Binäre Speicherung möglich
- Oft: Graue Metadaten separat zusammengefaßt

Aufteilung

Stream-Klasse (z.B. für Files)

- Ausgabe: Methoden
  - `writeInt()`, `writeString()` – kodiert nur Daten
  - `writeObject()` – schreibt zusätzliche Typinformation!
- Eingabe: Methoden
  - `readInt()`, `readString()` – kodiert nur Daten
  - `readObject()` – schreibt zusätzliche Typinformation!

Shape Objekte

- `read()` / `write()` definieren für jedes Shape
- Aufruf der Stream-Methoden

Vorteile der Aufteilung

Vorteile

- Verschiedene Streams (Netzwerk, Dateien)
- Einheitliches Dateiformat
- Typerkennung bei „`read()`“
  - Objekt der richtigen Klasse muss wieder angelegt werden
  - Mechanismus nur einmal implementiert
  - Informationen bei „write“ entsprechend anlegen
- Sicherheitsprüfungen
  - Erkennung eines Overruns bei read (Länge der Objekte nochmal speichern)
  - Erkennung unbekannter Typen (Fehlermeldung oder Auslassung/Recovery)

Weitere Herausforderungen

Änderungen ?!

- Dateiformat ist nun völlig von Datenlayout abhängig
- Widerspricht OO-Idee (Kapselung)

Maßnahmen

- Versionen
  - Für jede Klasse
  - Für jedes Attribut
- Testen, ob Attribut schon bekannt (Versionsvergleich)
  - Felder können einfach hinzugefügt werden (Version++)
- Komplexere Fälle
  - Spezielle Versionsabfrage in write/read für Kompatibilität

Weitere Herausforderungen

Zyklische Graphen von Objekten

- „`write()`“ schreibt auch enthaltene Objekte
- Zyklische Referenzen
  → Endlose Rekursion bei „`WriteObject()`“

Abhilfe

- (Hash-)Tabelle mit schon geschriebenen Objekten mitführen
  - Zyklen werden erkannt
  - Jedes Objekt wird nur einmal geschrieben
  - Danach lediglich ein Verweis auf die erste Kopie

Ähnlich

Kopieren von Objekten

- „`assign()`“, „`copy()`“, „`operator=()`“, copy-Konstruktor
- Im Prinzip gleicher Mechanismus

Netzwerktransfer

- Spezielle Streams

Synchronisation, z.B.

- Datenbank
- Redundanter Rechner (Ausfallsicherheit)
- Front-End ↔ Back-End

Standard OOP Design:

Reflection & Introspection

Zusammenfassung

Objekte

- Können sich darstellen & bearbeiten
  - z.B. „Shapes“
- Können sich speichern & laden („Serialisierung“)
- Können sich kopieren, synchronisieren, etc.

Probleme

- Viel redundante Handarbeit
  - Fehler und Inkonsistenzen möglich
- Erweiterung des Mechanismus schwierig
  - *Neuer Versionierungsmech.:* Alle Methoden neu schreiben

Lösung: Gar nichts schreiben!

(Dynamische) Meta-Programmierung via Reflektion

- „Reflektion“ („reflection“) erlaubt, die Struktur der Klassen zur Laufzeit anzusehen
  - Auch bekannt als „Introspection“
- Verfügbar in SmallTalk, Python, JAVA, voraus. C++20
- Serialisierung kann damit vollständig automatisiert werden
  - Und noch mehr

Struktur (alle Sprachen)

Meta-Klassen

- „Meta-Klassen“ beschreiben Klassen *(Typinformation)*
  - In SmallTalk, Python: wörtlich; Klassen sind Instanzen von Meta-Klassen
  - In JAVA: Nur Beschreibung
    - „Reifikation“: Compiler spiegelt Code-Informationen in Datenstruktur, die zur Laufzeit verfügbar ist
  - In C++: Arbeit das Std-Committee an der Definition
    - In „unserem eigenen“ GeoX: Selbstgebaut

Struktur (alle Sprachen)

Kurz: Meta-Klasse = Laufzeitersatz für Klasse

- Jede Objektinstanz kann einer Meta-Klasse zugeordnet werden (die ihre Klasse beschreibt)
- Felder, Methoden der Klassen können erfragt werden
- Meta-Klassen können neue Objekte vom repräsentierten Typ anlegen

Reflection in Python

Sehr einfach – alles sind Objekte

- `type(obj)` – Gibt das Klassenobjekt zurück
  - `type(type(obj)) = <class 'type'>`
  - Klassenobjekt ist (in der Regel) Instanz von „`type`“
  - „`type`“ ist die Standard-Meta-Klasse
- `m = getattr(obj, "Name")` – Zeiger auf Member holen (Nachschl. nach Strings)
- `m = setattr(obj, "Name", value)` – Wert setzen
- `callable(m)` – Prüft, ob ein Attribut aufgerufen werden kann (Methode?)
- `m(param1, param2)` – Meth.-Aufr. (`self` in `m` gebunden)

Automatische Serialisierung

Methoden „write“ / „read“

- Erfragen alle Eigenschaften der Klasse
- Schreiben/lesen diese in/von Datei
- Inklusive Verweise auf andere Objekte
  - Mit „`write`/`readObject()`“ der Stream-Klasse
  - Automatische Auflösung von Zyklen

Standardbibliotheken

- In Python: „`Pickle`“-Packet
- In Java: „`Serialization`“ (Standard)

Umstritten?

Nachteile 1: Versionierung

- Mehraufwand für Versionierung nötig
- „Abbildung“ zwischen Versionen
  - Alte auf neue Felder abbilden oder umgekehrt
  - Ggf. komplexere Transformation des Objektgraphens bei komplexeren Änderungen
- Definition der Inter-Versions-Abbildungen
  - Per Attributnamen + Defaultvalue (unflexibel, sehr einfach)
  - Mit „Mapper“-Funktionsobjekten
    - In der neusten Klassenversion für Laden alter Dateien
    - In der Datei für Laden neuer Dateien mit alter Klasse
- Meine Einschätzung: Lösbar, aber gewisser Aufwand

Umstritten?

Nachteile 2: Sicherheitsrisiken

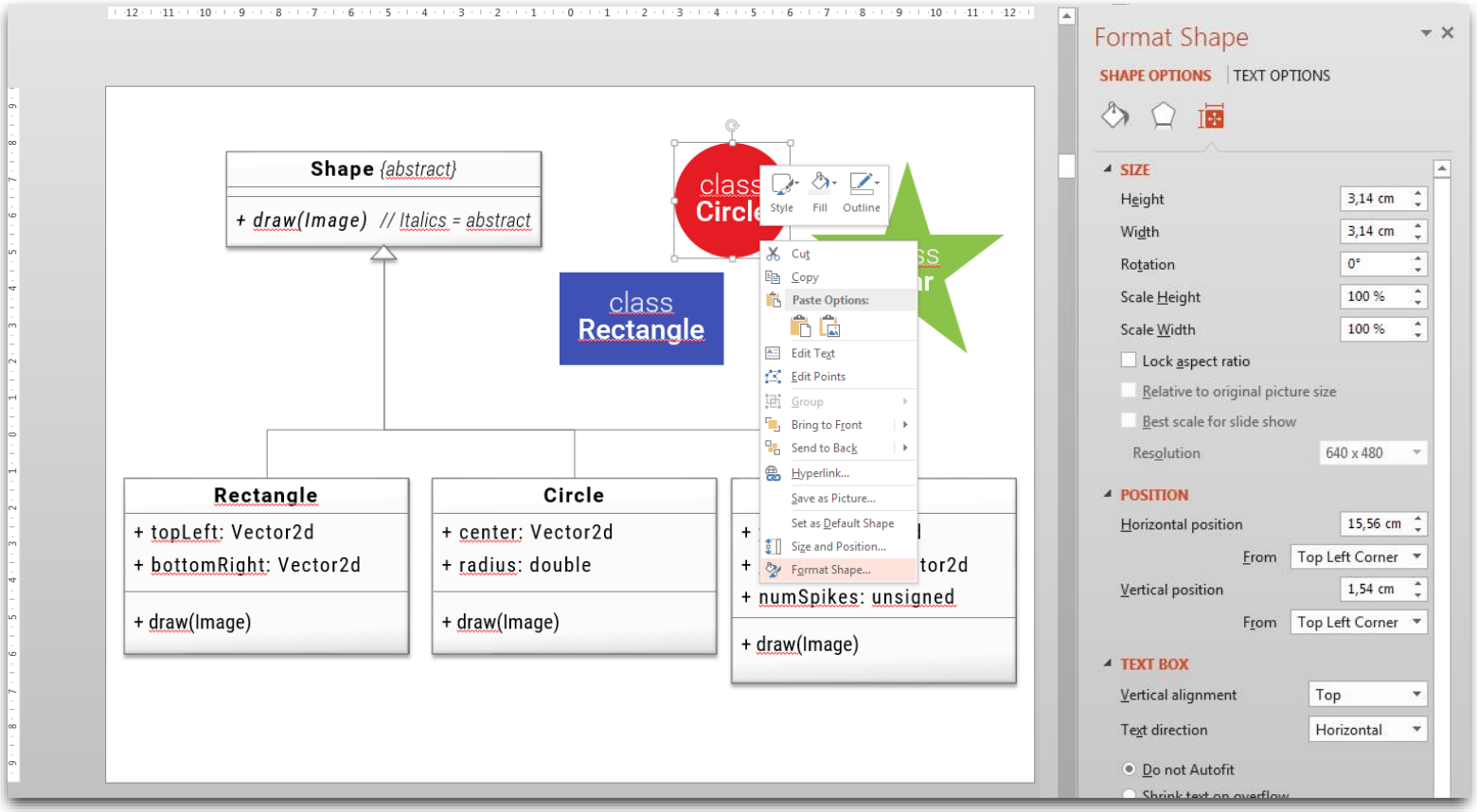
- Bösartige Kommunikationsteilnehmer!
  - z.B. Objekte über Internet senden
  - Von Front-End zu Back-End
  - Manipulierte Dateien (Viren im Attachment)
- Direktes Schreiben der Attribute möglich
  - Inkonsistente Zustände
  - Kontostand = 10.000.000€ (why not?)
- Aufwendige Prüfungsmechanismen nötig
- „Pickle“-Doku warnt z.B. davor:
  - Nicht sicher bei bösartigen Nutzern!

Was stattdessen?

„More Sophisticated“

- Abbildung auf spezielle „persistente“ Darstellung
- Eigenes Datenformat dafür definieren
  - Kann von Implementation stärker abstrahieren
  - Unter Sicherheitsaspekten definieren
- Weitgehende Automatisierung weiterhin möglich
  - Sogar nötig? Fehler vermeiden?
  - Immer mehr Aufwand nötig für sichere Protokolle!

Andere Anwendungen



Anwendungen von Reflection

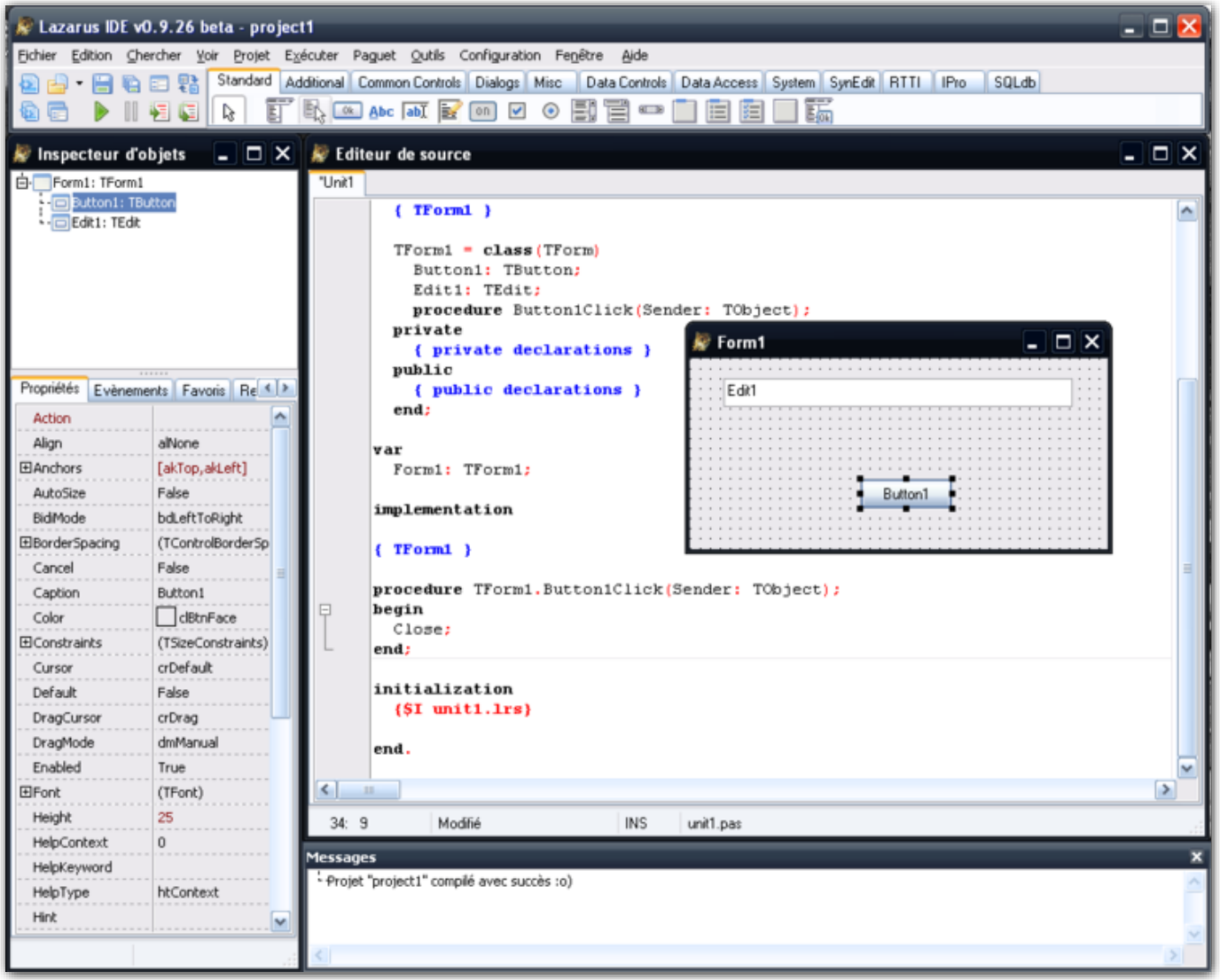
- (Einfache) GUIs automatisch bauen
- Property Inspector (z.B. NextStep, QT, Delphi)

Grundidee

Introspection

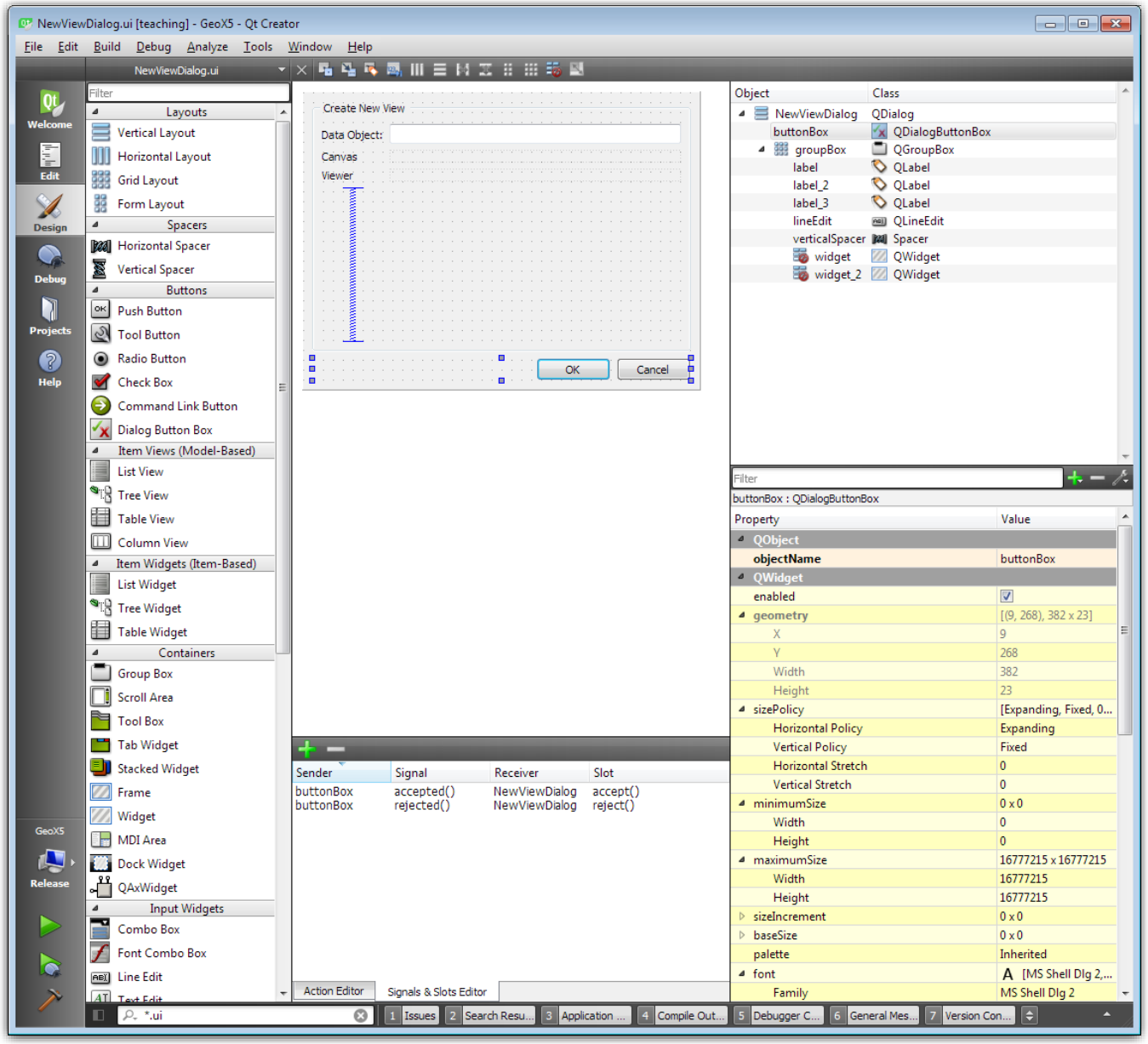
- Editor für ein Objekt bauen
  - Bestimme alle Felder der Klasse
  - Prüfe, ob „öffentlich“ für GUI
    - Ggf. Entsprechende Annotation nötig (Python: z.B. Annotations, Dekorators)
  - Erzeugen eines GUI-Elements für das Feld
  - Einige generische Typen (keine komplexen GUIs)
- Wenn Editor läuft
  - Schreiben / Lesen der Werte GUI ↔ Object
  - Introspection / Meta-Klassen für Zugriff auf Felder

Beispiel: Delphi / Lazarus

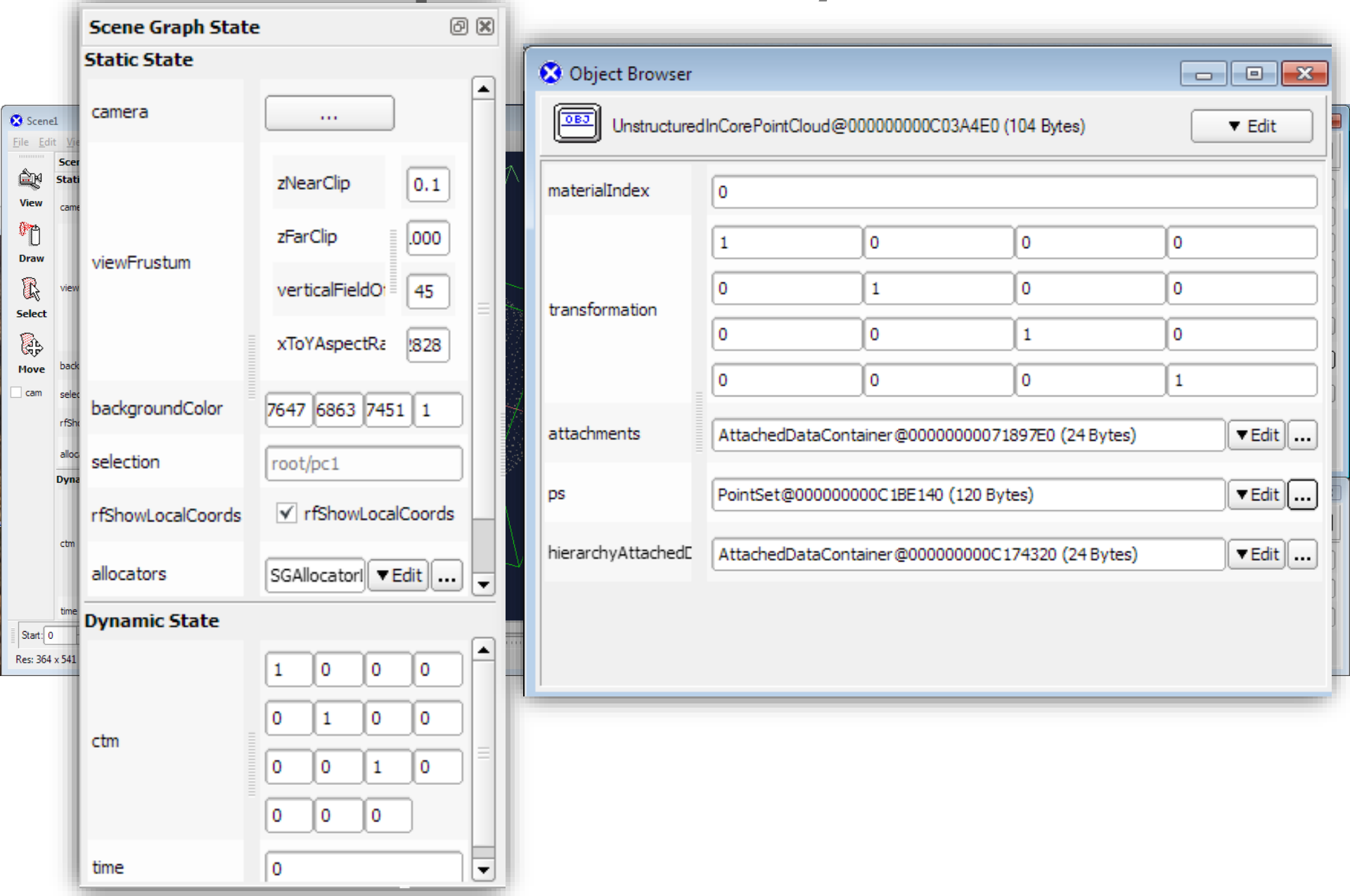


https://commons.wikimedia.org/wiki/File:Lazarus_IDE_9-26.png

Beispiel: QT


NewViewDialog.ui [teaching] - GeoX5 - Qt Creator
File Edit Build Debug Analyze Tools Window Help
Welcome
Edit
Design
Debug
Projects
Help
GeoX5
Release
Filter
Layouts
Vertical Layout
Horizontal Layout
Grid Layout
Form Layout
Spacers
Horizontal Spacer
Vertical Spacer
Buttons
Push Button
Tool Button
Radio Button
Check Box
Command Link Button
Dialog Button Box
Item Views (Model-Based)
List View
Tree View
Table View
Column View
Item Widgets (Item-Based)
List Widget
Tree Widget
Table Widget
Containers
Group Box
Scroll Area
Tool Box
Tab Widget
Stacked Widget
Frame
Widget
MDI Area
Dock Widget
QAxWidget
Input Widgets
Combo Box
Font Combo Box
Line Edit
Create New View
Data Object:
Canvas
Viewer
OK
Cancel
Object
Class
NewViewDialog
QDialog
buttonBox
QDialogButtonBox
groupBox
QGroupBox
label
QLabel
label_2
QLabel
label_3
QLabel
lineEdit
QLineEdit
verticalSpacer
Spacer
widget
QWidget
widget_2
QWidget
buttonBox : QDialogButtonBox
Property
Value
QObject
objectName
buttonBox
QWidget
enabled
geometry
[(9, 268), 382 x 23]
X
Y
Width
Height
sizePolicy
[Expanding, Fixed, 0...
Horizontal Policy
Expanding
Vertical Policy
Fixed
Horizontal Stretch
0
Vertical Stretch
0
minimumSize
0 x 0
Width
0
Height
0
maximumSize
16777215 x 16777215
Width
16777215
Height
16777215
sizeIncrement
0 x 0
baseSize
0 x 0
palette
Inherited
font
[MS Shell Dlg 2,...
Family
MS Shell Dlg 2
Sender
Signal
Receiver
Slot
buttonBox
accepted()
NewViewDialog
accept()
buttonBox
rejected()
NewViewDialog
reject()
Action Editor
Signals & Slots Editor
*.ui
1 Issues
2 Search Resu...
3 Application ...
4 Compile Out...
5 Debugger C...
6 General Mes...
7 Version Con...

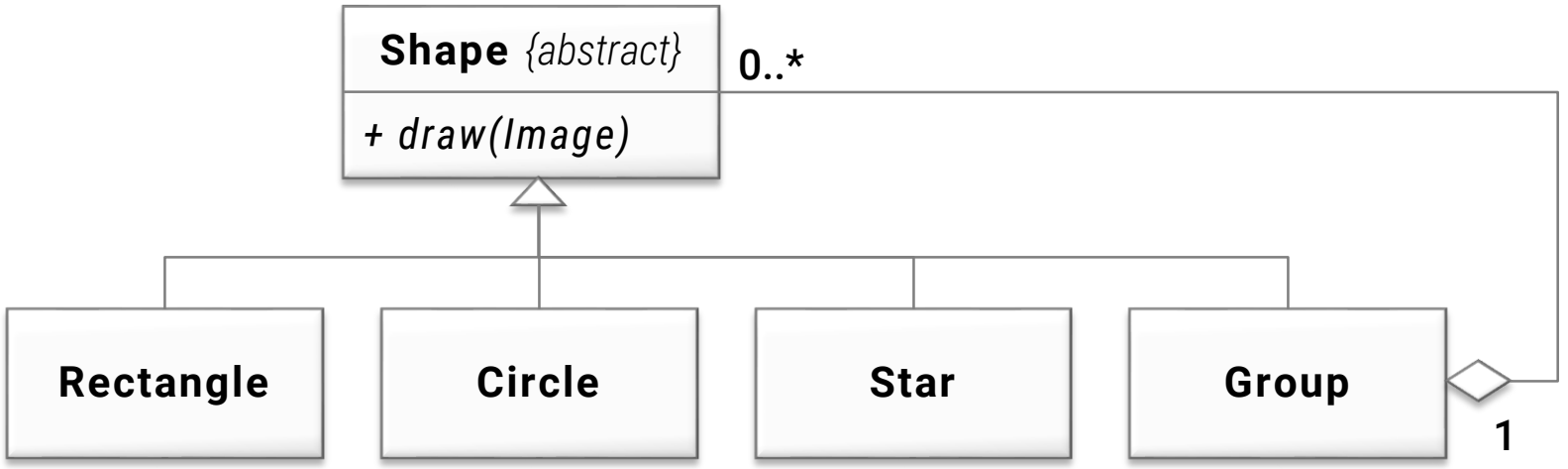
Beispiele: GeoX / GeoXL



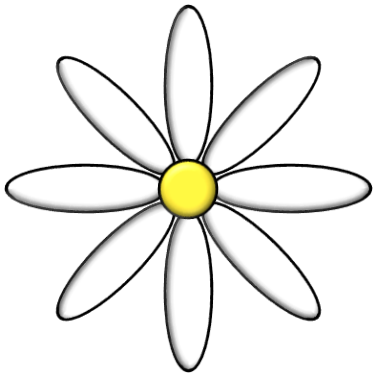
Funktionale Variante:

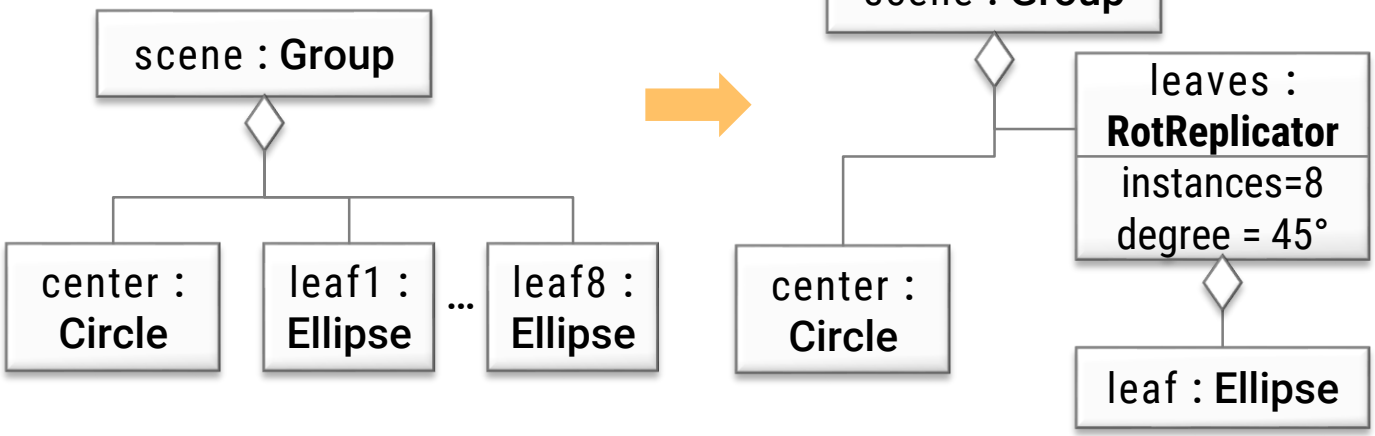
„Datenflussgraphen“

Komponentenhierarchie



Feature: Programmable Repetition?

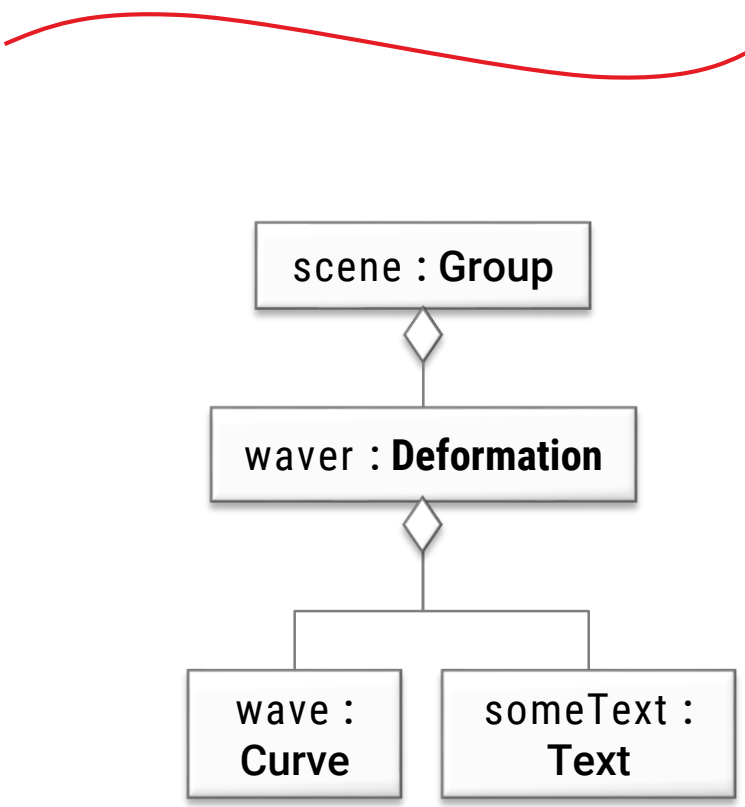




Anderes Bespiel

Wavy-Text!

+



Wavy-Text!

Reifikation von „draw()“-Befehlen

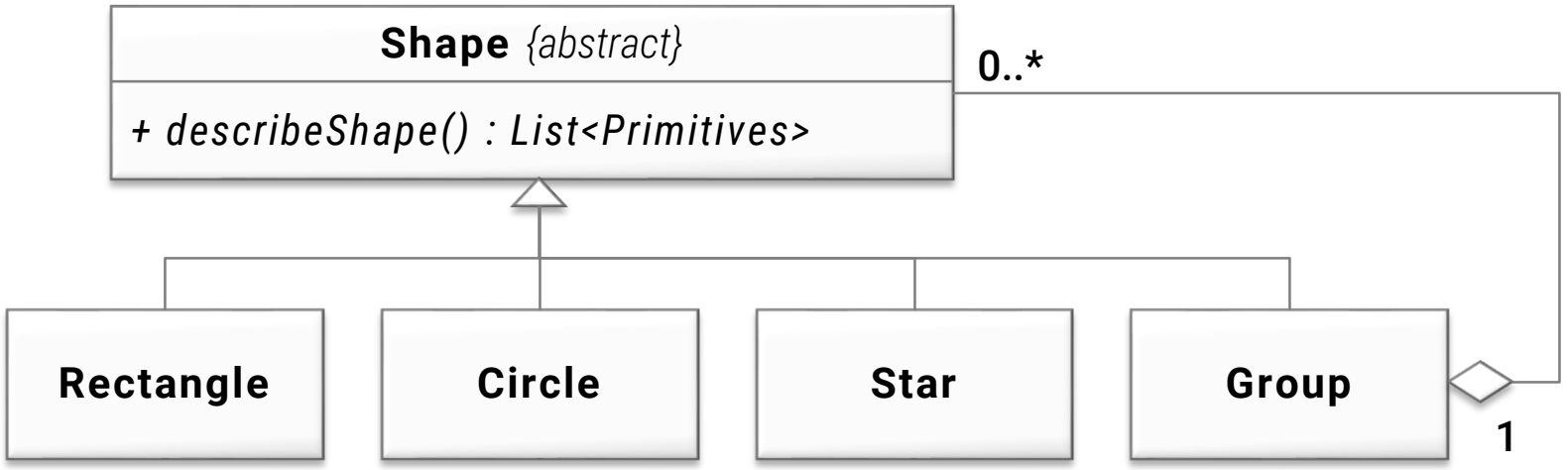
Limitierung

- Zeichnen via „`draw(Image *img)`“, z.B.
  - `img->drawEllpise(center, radius1, radius2);`
- Methodenaufrufe kann man wiederholen, aber nicht verändern

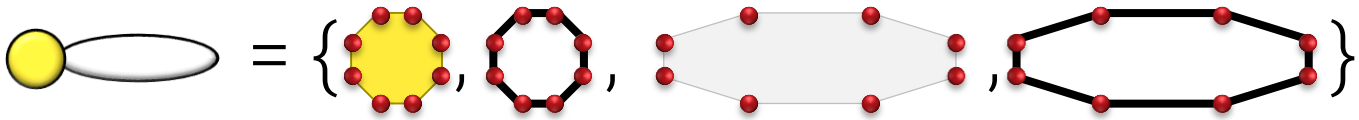
Reifikation

- Liste von Datenobjekten
- Elementare „Primitive“: Linien, Polygone

Komponentenhierarchie



Repräsentation: Liste von Primitiven



Flexibler

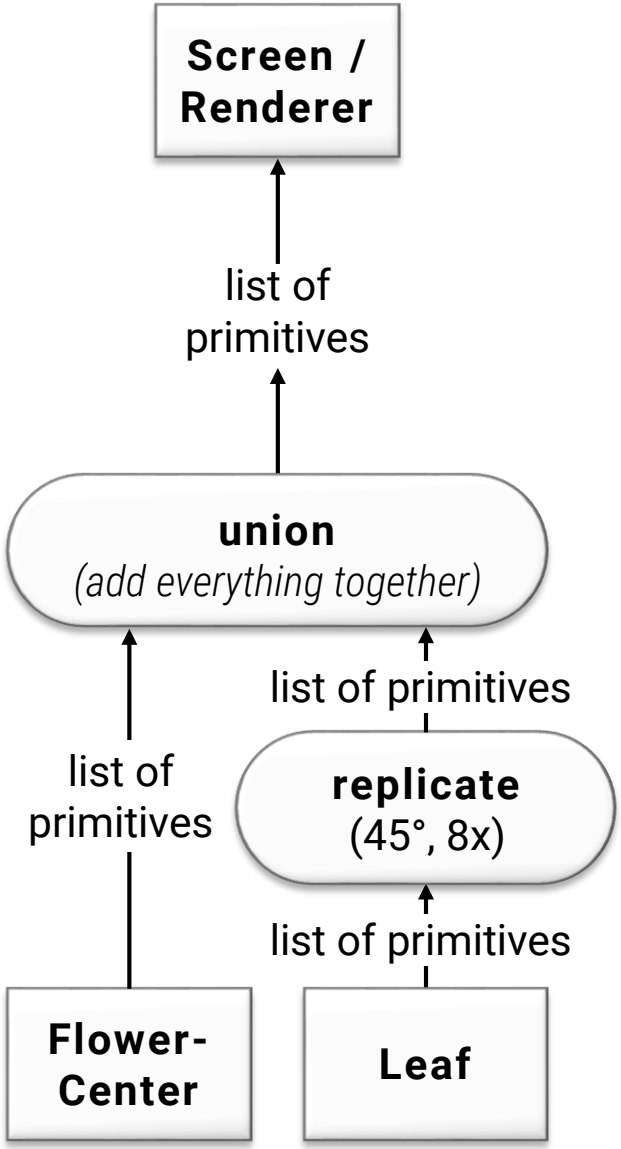
Nun können wir unser Problem lösen

- Jeder Knoten im Objektbaum kann eine geometrische Beschreibung
  - Aufruf „describeShape()“ des Kindkonten
  - Liste von Primitiven (zuvor: Operationen in „`draw()`“)
- Beliebige Berechnungen damit möglich
- Zusammenstellen einer neuen Liste

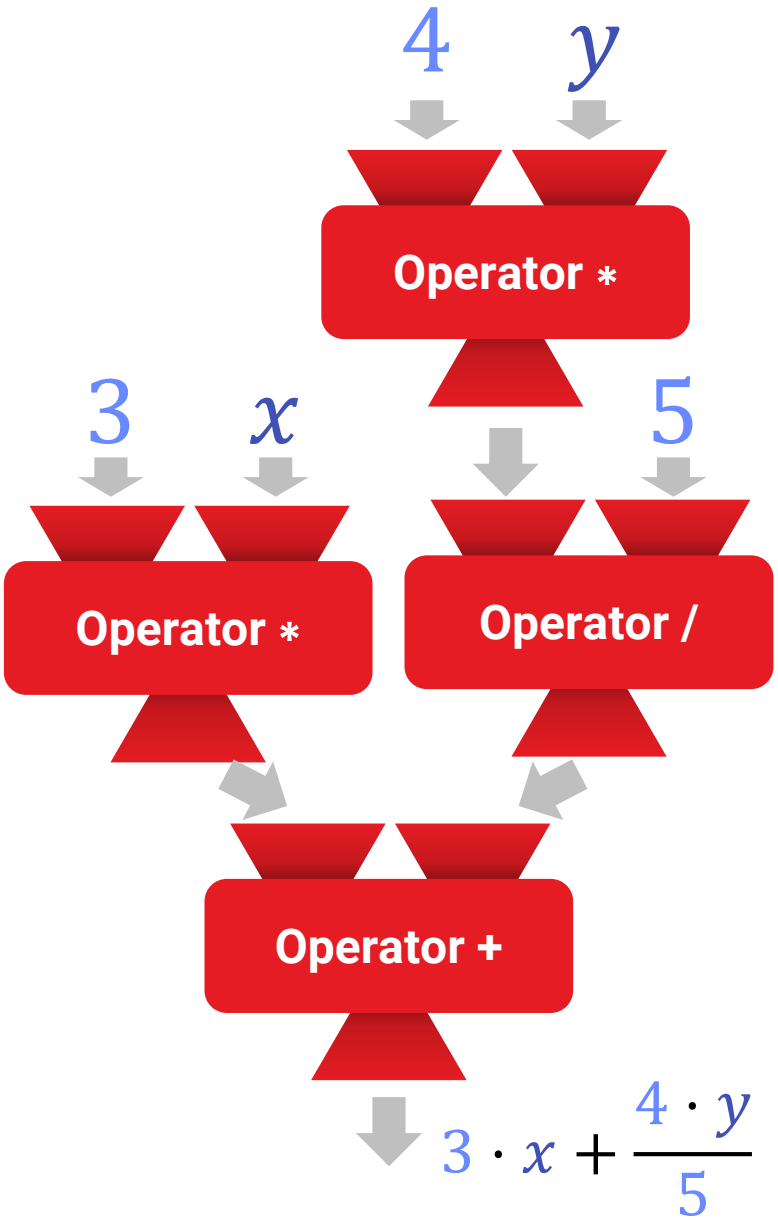
Datenflussgraph

Architektur

- Knotenobjekte sind „Funktionen“
  - 0…* Eingaben
  - 0…* Ausgaben
  - Beliebige berechnung, um Eingaben in Ausgaben zu transformieren
- Jedes Knotenobjekt repräsentiert eine Funktion
- „Funktionales“ Muster: Kompositionshierarchie/Graph
- Oft in der Praxis angewandt

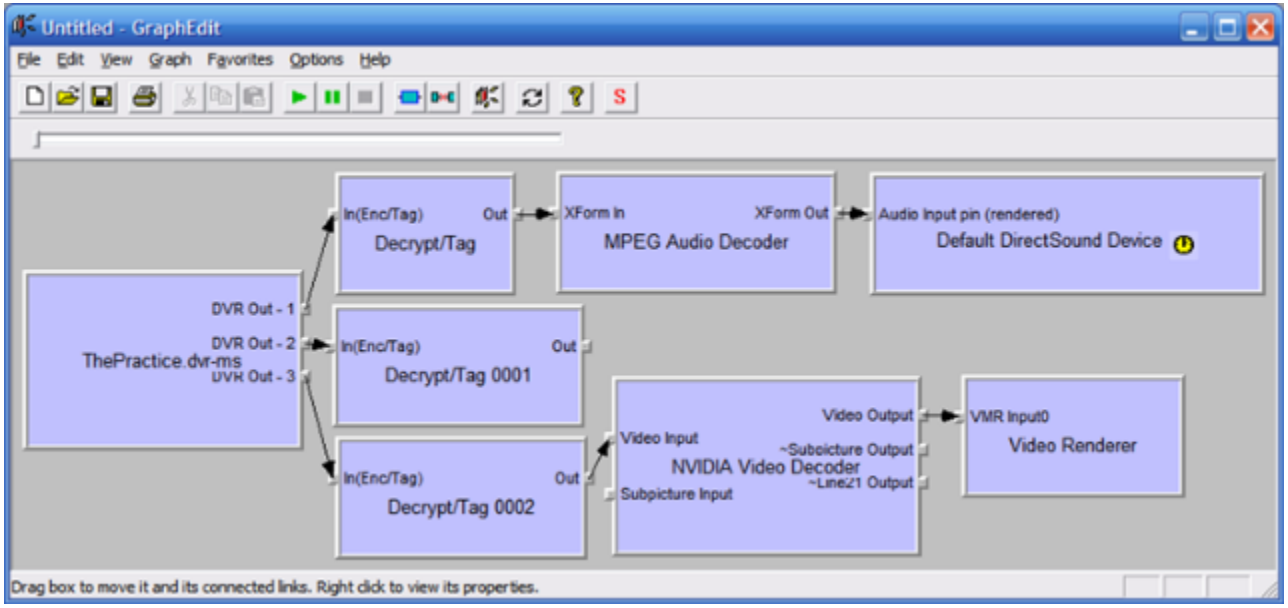


Einfache Funktionen



“$3 \cdot x + \frac{4 \cdot y}{5}$”

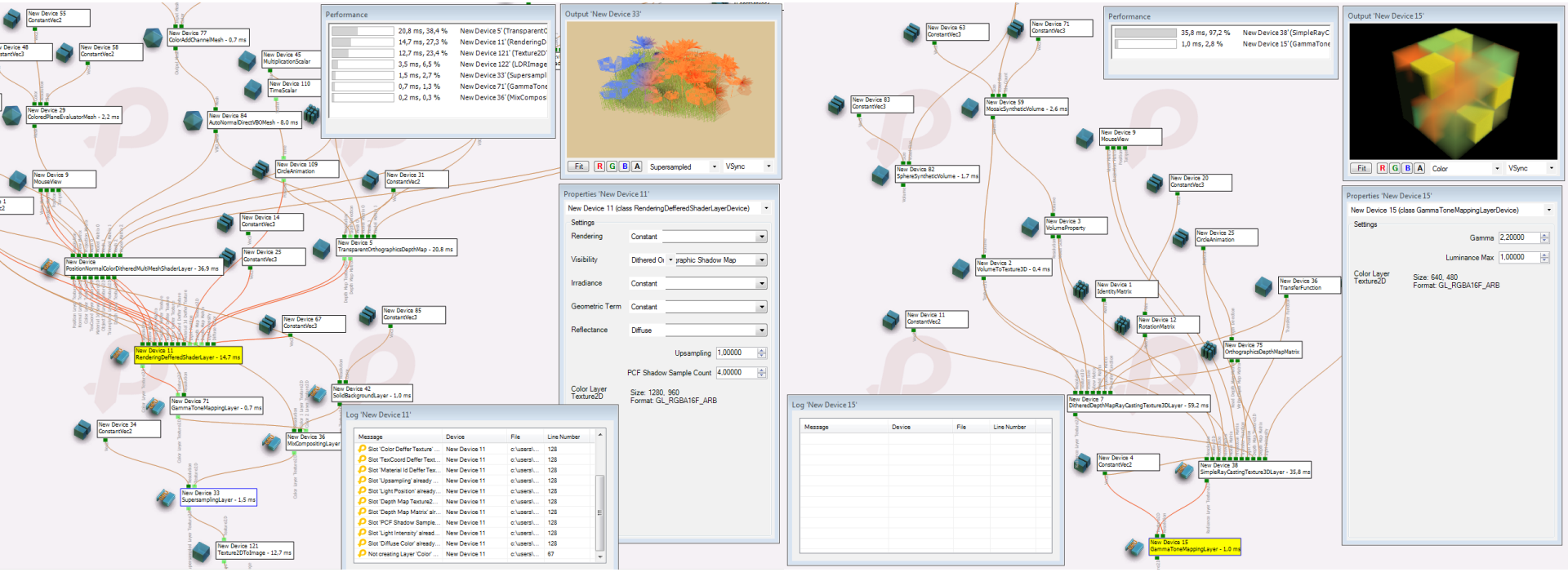
Beispiele



Microsoft „DirectShow“

- Graph von Audio / Video Prozessoren
- Data flowing: Bild-, Video-, Audio-Puffer

Beispiele



„**Plexus**“ (Tobias Ritschel, UCL/MPI Saarbrücken)

- 2D Bildverarbeitung und 3D Rendering
- Daten werden mit der GPU verarbeitet

Beispiele

Datenflussgraphen

- „DirectShow“ Graphen (Microsoft DirectX Framework)
- Plexus Framework (Tobias Ritschel, UCL/MPI Informatik)
- Amira Visualisierungssoftware
  (ZIB Berlin / Thermo Fischer Scientific)
- TensorFlow (Deep Learning Framwork von Google)

“OOP”, “Functional”, und das “Expression Problem”

Zwei Varianten

Objekt-Orientierte Variante

- Oberklasse mit festen Methoden (z.B. „draw()“)
- Neue Datentypen durch Ableiten
  - Möglich, ohne Oberklasse zu ändern/neu zu übersetzen
- Methoden erweitern ist aufwendig

Funktionale Variante

- Fester Satz an Datentypen (z.B. Primitive)
- Neue Operationen durch hinzufügen von Funktionen
  - „Funktionsobjekte“ in einer OOP-Sprache
- Datentypen erweitern ist aufwendig

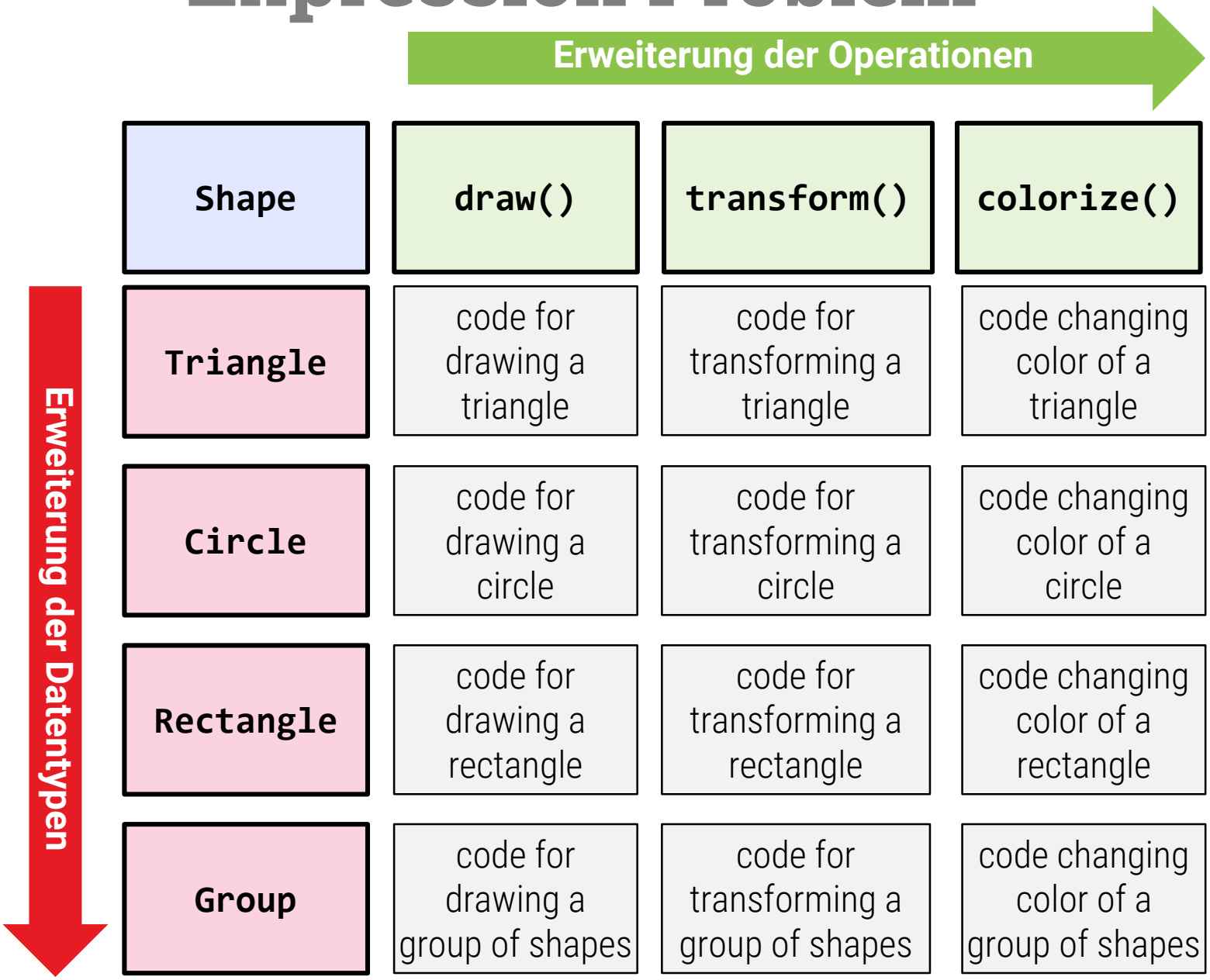
Funktionale Sprache

```
// Pseudo-Code – so ähnlich in ML-Dialekt, Haskell o.ä.
type Shape = Triangle or Rectangle or Circle or Group
type Triangle = Vector2d p1, Vector2d p2, Vector2d p3
type Circle = Vector2d p1, double radius
// ...

fun draw(Shape s, Graphics g) : Graphics
  switch typeof(s)
  case Triangle:
     // create new g with triangle added
     return g_result
  case Rectangle:
     // create new g with triangle painted on it
     return g_result
  // ..others... Type system will check that all cases are handled!
  // Nonetheless, we need to modify all functions if we add types
end fun
```

C++ Variante (unchecked)

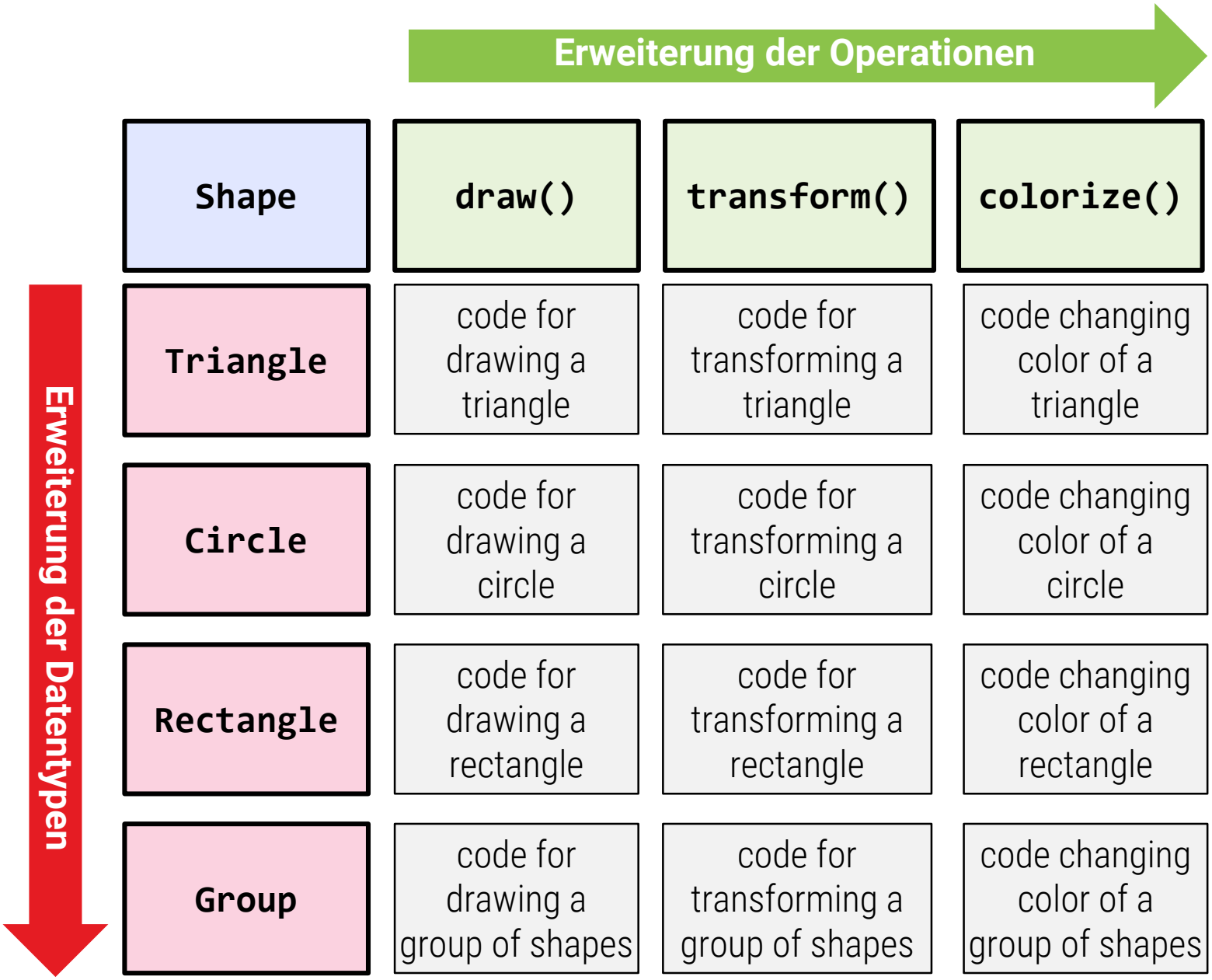
```
enum ShapeType {Type_Triangle, Type_Rectangle, Type_Circle, Type_Group};
struct Triangle { Vector2d p1; Vector2d p2; Vector2d p3;};
struct Circle {Vector2d p1; double radius;};
// ...
struct Shape {
   ShapeType type;
   union {
      Triangle tri;
      Circle circ;
      // ...
   };
};

void draw(const Shape s, Graphics &g) {
  switch (s.type) {
    case Triangle: {g.drawTriangle(s.tri); break;}
    case Circle: {g.drawCircle(s.circ); break;}
    // ..others... – not checked by compiler
  }
}
```

Expression Problem



Leicht in funktionalem Design, schwerer in OOP

Leicht in OOP-Design, schwerer in funktionalem



Shape	draw()	transform()	colorize()
Triangle	code for drawing a triangle	code for transforming a triangle	code changing color of a triangle
Circle	code for drawing a circle	code for transforming a circle	code changing color of a circle
Rectangle	code for drawing a rectangle	code for transforming a rectangle	code changing color of a rectangle
Group	code for drawing a group of shapes	code for transforming a group of shapes	code changing color of a group of shapes

OOP

Oberklasse ändern, alle Nachfahren anpassen, alles neu kompilieren, Bibliothekennicht mehr kompatibel, Source-Code der Ober-Kl. nötig

Klasse ableiten, implementieren

Shape	draw()	transform()	colorize()
Triangle	code for drawing a triangle	code for transforming a triangle	code changing color of a triangle
Circle	code for drawing a circle	code for transforming a circle	code changing color of a circle
Rectangle	code for drawing a rectangle	code for transforming a rectangle	code changing color of a rectangle
Group	code for drawing a group of shapes	code for transforming a group of shapes	code changing color of a group of shapes

Func

Neue Funktion schreiben, alle Typen berücksichtigen

alle Funktionen durchgehen und Implementation hinzufügen
Source-Code nötig, nicht mehr kompatibel

Shape	draw()	transform()	colorize()
Triangle	code for drawing a triangle	code for transforming a triangle	code changing color of a triangle
Circle	code for drawing a circle	code for transforming a circle	code changing color of a circle
Rectangle	code for drawing a rectangle	code for transforming a rectangle	code changing color of a rectangle
Group	code for drawing a group of shapes	code for transforming a group of shapes	code changing color of a group of shapes

Lösungen für das Expression-Problem

Beispiele für systematische Lösungen

- „Visitor“-Pattern in OOP (kehrt Problem für OOP um)
- Offene Klassen
  - Methoden nicht Teil des Typs (z.B. in Closure-“Multi“)
- Patching
  - Methoden Teil des Types, der selbst dynamisch änderbar ist
  - Methoden dynamisch hinzufügen in Python („Monkey-Patching“)
  - Extension-Methods in C#
- Type-Classes u.ä. in Haskell

Höhere Komplexität

- In praktischen Anwendungen eher vermieden

Visitor Pattern

```
class ShapeVisitor {
 public:
   virtual void visitTriangle(Triangle *t) = 0;
   virtual void visitRectangle(Rectangle *t) = 0;
   virtual void visitCircle(Circle *t) = 0;
   virtual void visitGroup(Group *t) = 0;
};

class DrawShape : public ShapeVisitor {Graphics g; ...};
class TransformShape : public ShapeVisitor {Transformation t; ...};
// ...extend as you like... (but adding types now becomes inflexible)

void visitListOfShapes(std::vector<Shape*> sv, ShapeVisitor *v) {
  for (Shape* s: v) {
  if dynamic_cast<Triangle*>(s) {v.visitTriangle((Triangle*)s);}
  else if dynamic_cast<Circle*>(s) {v.visitCircle((Circle*)s);}
  else if ... // all cases
  else std::cout << "too bad - we cannot extend types easily anymore now";
}
```

Dynamic Dispatch in Functional

Bei funktionalen Sprachen / Design

- Erweiterung von Datentypen mit Funktionszeigern
- „VMTs“ nachgebaut
- Damit auch OOP in funktionalem Design möglich

Alles das gleiche?

- Jeweils mehr/weniger Schreibarbeit
- Hauptunterschied (zusätzlich zum gesagten)
  - Vermeiden von mutable state [global zu betr. Code]
  - Datenfluss statt Mutation
  - Oft (nicht immer) leichter zu verstehen