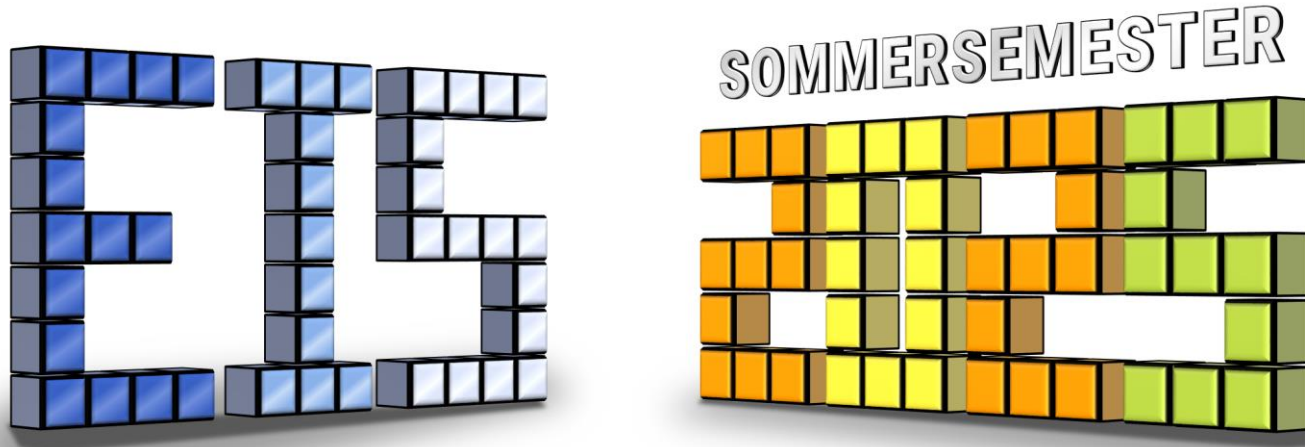


# EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #12

## Fortgeschrittene Programmiertechniken

Michael Wand  
**Institut für Informatik**  
[Michael.Wand@uni-mainz.de](mailto:Michael.Wand@uni-mainz.de)



# Techniken

## Strukturierung von Programmen

- Prozedural

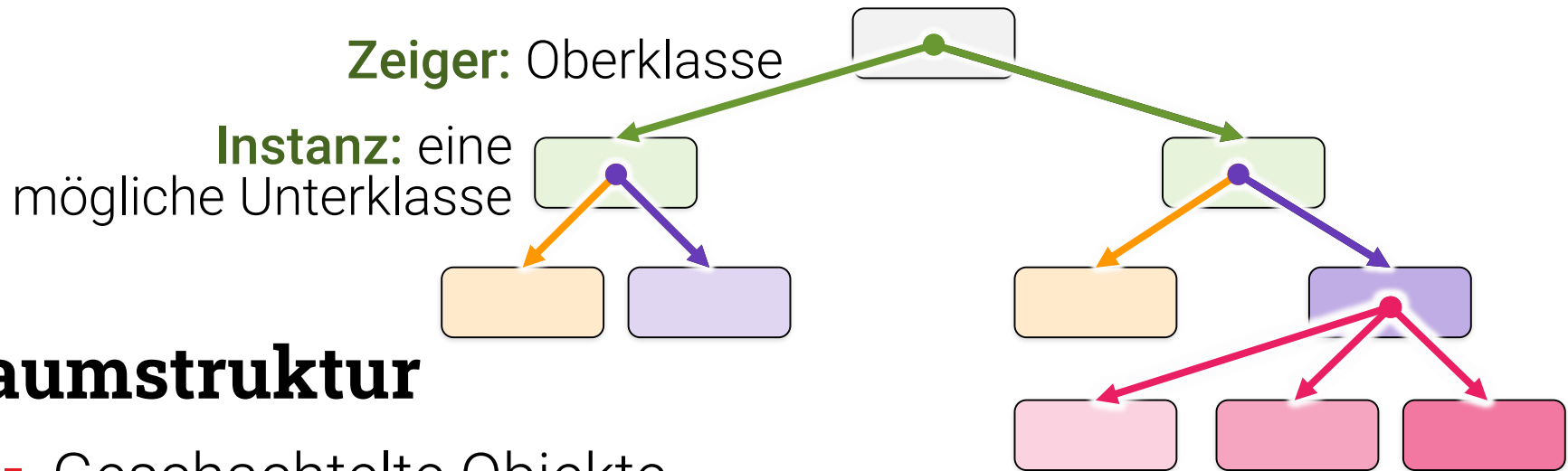
- Objekt-orientiert
- Funktional
- Meta-Programmierung

## Heute: „Schritt zwei“

- Nicht die „Kernideen“
- Aber sehr nützlich, wenn man die Basics kennt

**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

# Ohne GC – Speichermanagement

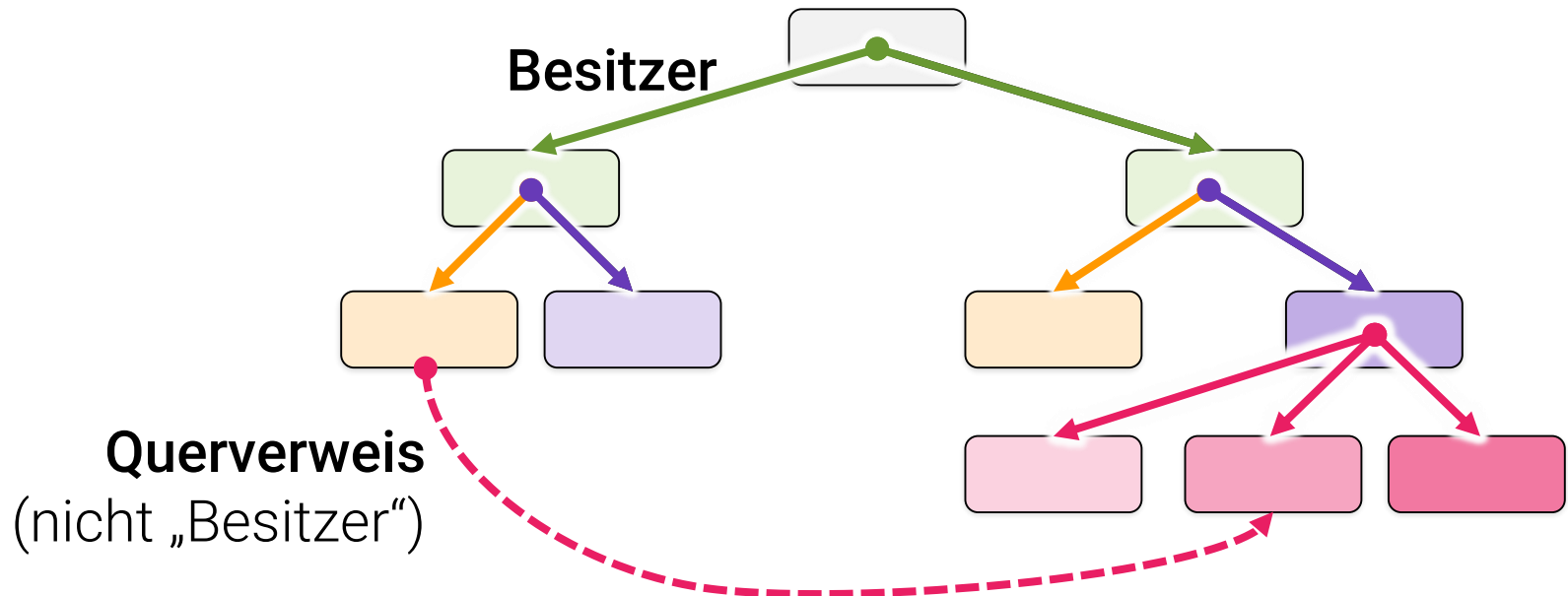
## In C++ & Co: manuelle Speicherfreigabe

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

## In Python, JAVA, Scala

- Immer noch nützlich zu wissen, wem was gehört
- GC löscht nur unerreichbare Objekte...

# „Parents are Owners“-Ansatz



## Konsistenz bei Querverweisen

- Baumstruktur ist automatisch konsistent
- Bei Querverweisen „dangling 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)

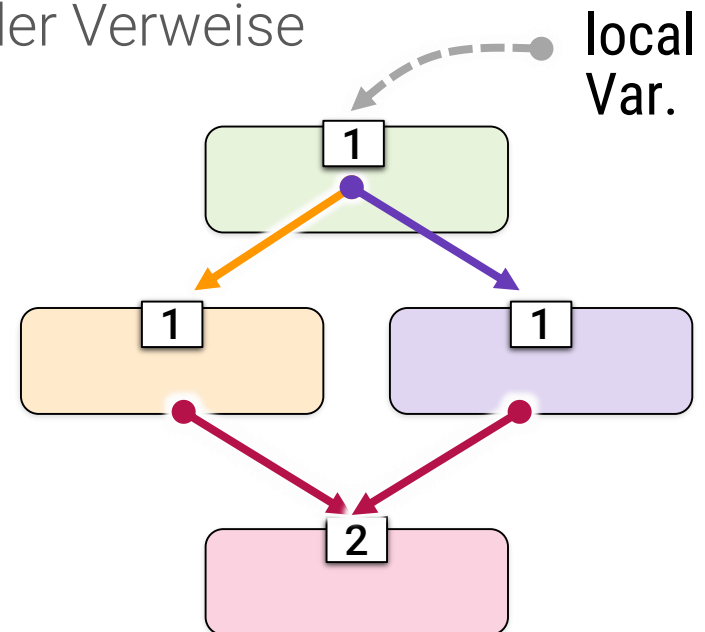
## Beispiel

- Qt Signals & Slots:  
Destruktor entfernt Objekte von Empfängerliste

# „Parents are Owners“-Ansatz

## Lösungsvorschlag 2: Reference-Counting

- Relevant bei manueller Speicherverwaltung (z.B. C++)
- 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

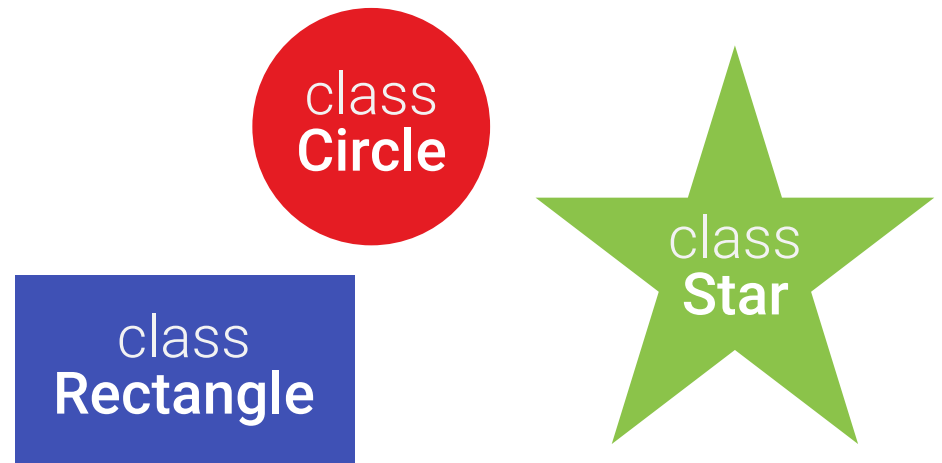
- Jedem Objekt lokal (in Bezug auf Parent) eindeutigen Namen geben
- Zugriff über Zeichenketten
  - Fehlerbehandlung, falls Name nicht gefunden
- Beispiele für symbolische Verweise
  - „/Dokument-A/Paragraph[1]/Zeichen[7]/Format“
  - „Dokument-A.Paragraph[1].Zeichen[7].Format“
- **Vorteil:** Sehr flexibel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

# Serialisierung & dynamische Metaprogrammierung

# Unvollständige Anwendung

## Was fehlt unserem *Vektorzeichenprogramm* noch?

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



# Evolution des Designs

## Speichern Prozedural

- Funktionen

```
„def save_document(d: Document, f: File)“  
„def load_document(f: File): Document“
```

- 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 isinstance(shape, Circle): ...  
elif isinstance(shape, Star): ...  
...
```

# Probleme Prozedural

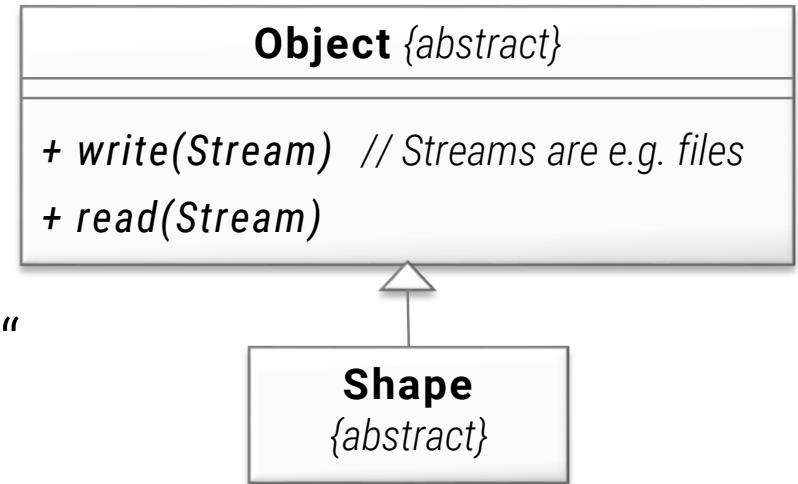
## Diverse Probleme

- Schlecht erweiterbar
- Neue Shapes:
  - Änderung des Dateiformates
  - Fallunterscheidungen müssen eingefügt werden
    - Nicht vergessen! (Exhaustivity Check hilft)
- Nach Übersetzung nicht mehr erweiterbar
  - Schlecht für Plug-Ins
- Dateiformat handdefiniert
  - Inkonsistenzen möglich
  - Rekonstruktion **save** → **load** im Fehlerfall nicht garantiert

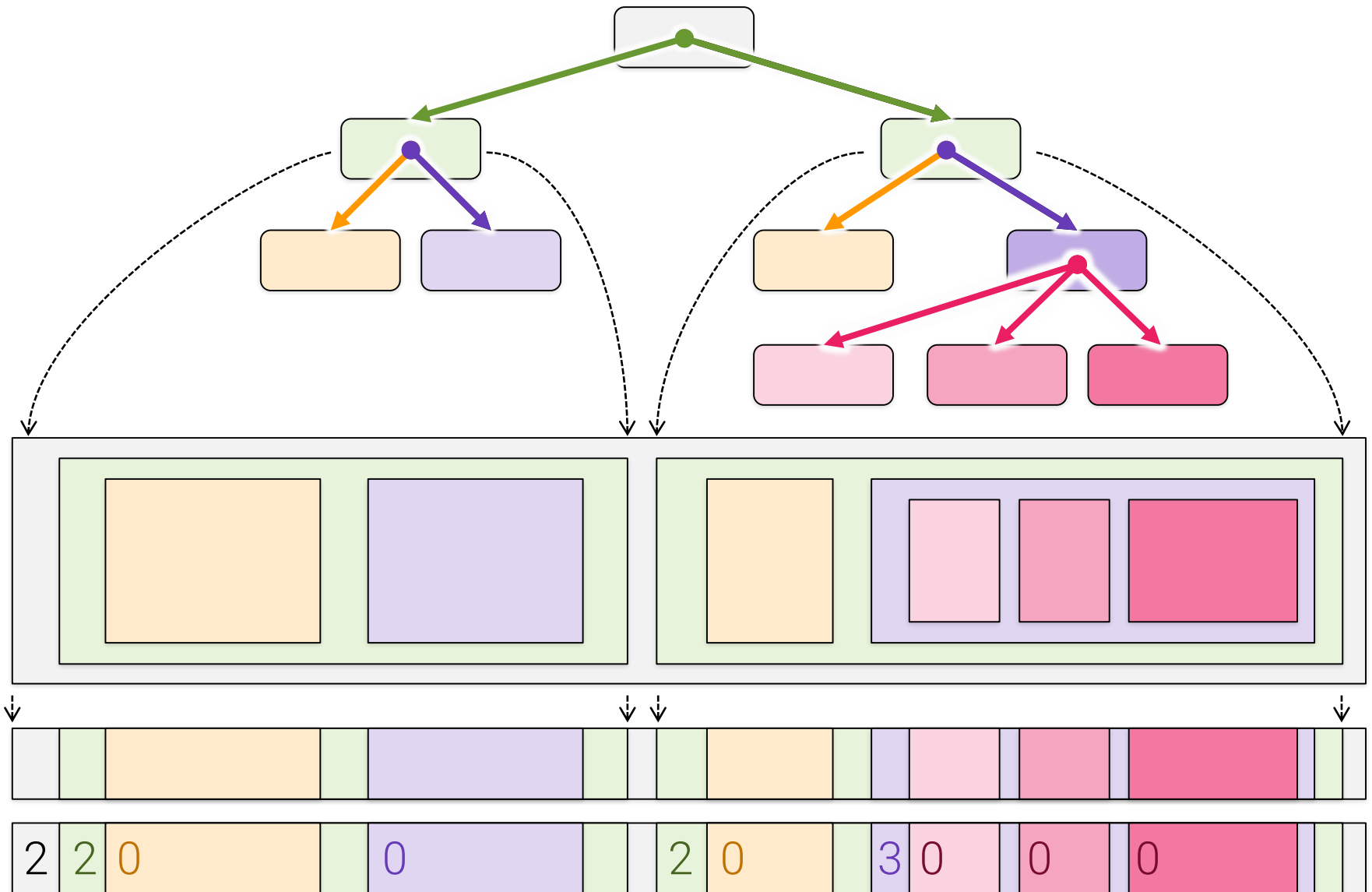
# Evolution des Designs

## Objektorientiert

- Member-Funktionen
  - „**void** **read**(**f**: **Stream**)“
  - „**void** **write**(**f**: **Stream**)“in Basisklasse!
- Methoden handgeschrieben
  - Schreibt alle Felder in Datei
  - Inkrementell erweitert in Nachfahrenklassen
- Konsistenz?
  - Jede Klasse: alle „ihre“ Felder lesen bzw. schreiben
  - Zusätzliche „Größenmarker“ um Fehler zu erkennen
    - Hier via hypothetischer „stream“ Hilfsklasse (Details folgen)



# „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 statt unstrukturierte Dateien

- 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: Ähnliche Probleme

- Python: `copy()` + `deep_copy()`
- Java: „`assign()`“, „`copy()`“,
- C++ „`operator=()`“, copy-Konstruktor

## 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++<sup>26</sup>
- 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
    - Type „type“ ist Standard Metaklasse
    - Eigene Typen möglich
  - 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
    - Meine Beispielapp „GeoX(L)“ (C++): selbstgebaut
    - Relativ rudimentäre Implementation auch in Qt (und, mehr oder weniger umfangreich, auch vielen andern Frameworks)

# Struktur (alle Sprachen)

## **Klassen beschreiben Struktur von Objekten**

- Metaklassen: Typ (Schablone) für Klassen

## **Was muss eine (Laufzeitrepräsentation) einer Klasse können?**

- Jede Objektinstanz kann einer Klasse zugeordnet werden
- 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 = obj.method`  
`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/read\_object()**“ der Stream-Klasse
  - Automatische Auflösung von Zyklen

## Standardbibliotheken

- In Python: „**Pickle**“-Packet (Standard)
  - Schreibt / liest einfach alle Felder (keine Versionierung)
- 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

- Böartige 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 böartigen 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!



# Fallstudie: Reflection-based Serialization in der Praxis

# Meine eigene Erfahrung

## Serialisierung in GeoX(L) [C++]

- Versionierung
  - Eigene Version für jede Klasse
  - Jedes Feld hat eine Version
- Halbkrementell
  - Version wird um 1 erhöht bei jeder Änderung
  - Hinzufügen von Feldern ohne Aufwand
    - Konstruktor initialisiert Objekte nach neustem Layout
    - Vorhandene Felder werden gelesen nach Version
  - Wegnehmen von Feldern erfordert Fallunterscheidung in `read()`

# Meine eigene Erfahrung

## Serialisierung in GeoX(L) [C++]

- Automatische Serialisierung
  - Ererbtes read() / write() arbeiten automatisch
  - Bei Problemen können sie durch handgeschriebenen Code ersetzt werden
    - Nur bei größeren Änderungen nötig:
      - Attribute entfernen
      - Klassen umbenennen (dafür Alias-Erkennung)
    - Fehler (Overruns/Underruns) werden erkannt
    - Fehlende Attribute / Exceptions können behandelt werden
  - Kompatibilität von alten Dateien mit neuen Programmversionen
    - Aber nicht umgekehrt! (Fehlermeldung)

# Meine eigene Erfahrung

## Serialisierung in GeoX(L) [C++]

- Wie liefs?
  - Nutzung im akademischen Umfeld
  - Alle Nutzer/innen auf der neusten Code-Basis
  - Zuverlässige Kompatibilität – Stabiles Dateiformat über 20 Jahre
- Probleme
  - Häufigster Fehler:
    - Inkrementieren der Version vergessen
    - Registrierung von Feldern vergessen
      - (In C++ leider manuell nötig)
    - Streamingsystem warnt davor, keine Stabilitätsproblem

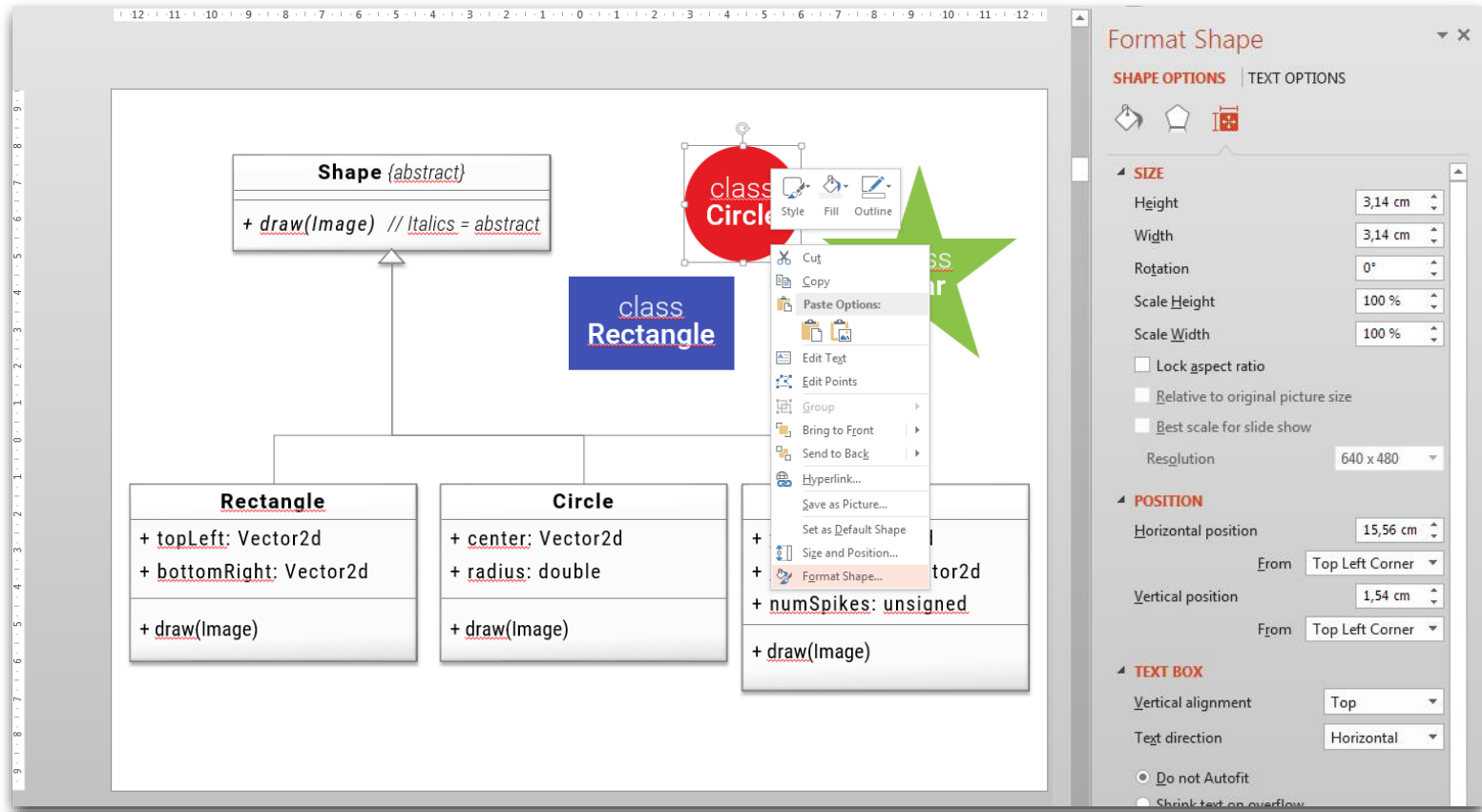
# Meine eigene Erfahrung

## Serialisierung in GeoX(L) [C++]

- Weitere Probleme
  - Ein schwerwiegendes Stabilitätsproblem
    - Ein schweres Problem: Dateiformatfehler bei Umstellung auf 64 Bit Speichermodell
    - Erforderte neue Dateiversion um Kompatibilität zu bewahren
- Keine Sicherheit
  - Das System ist leicht angreifbar
  - Für öffentliche APIs im Netzwerk nicht sinnvoll / zu riskant
  - Nur für „wohlwollende Benutzer“
  - Kein Problem für uns, da nie breit öffentlich genutzt

Was kann man noch alles  
mit Reflection machen?

# Andere Anwendungen



## Anwendungen von Reflection

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

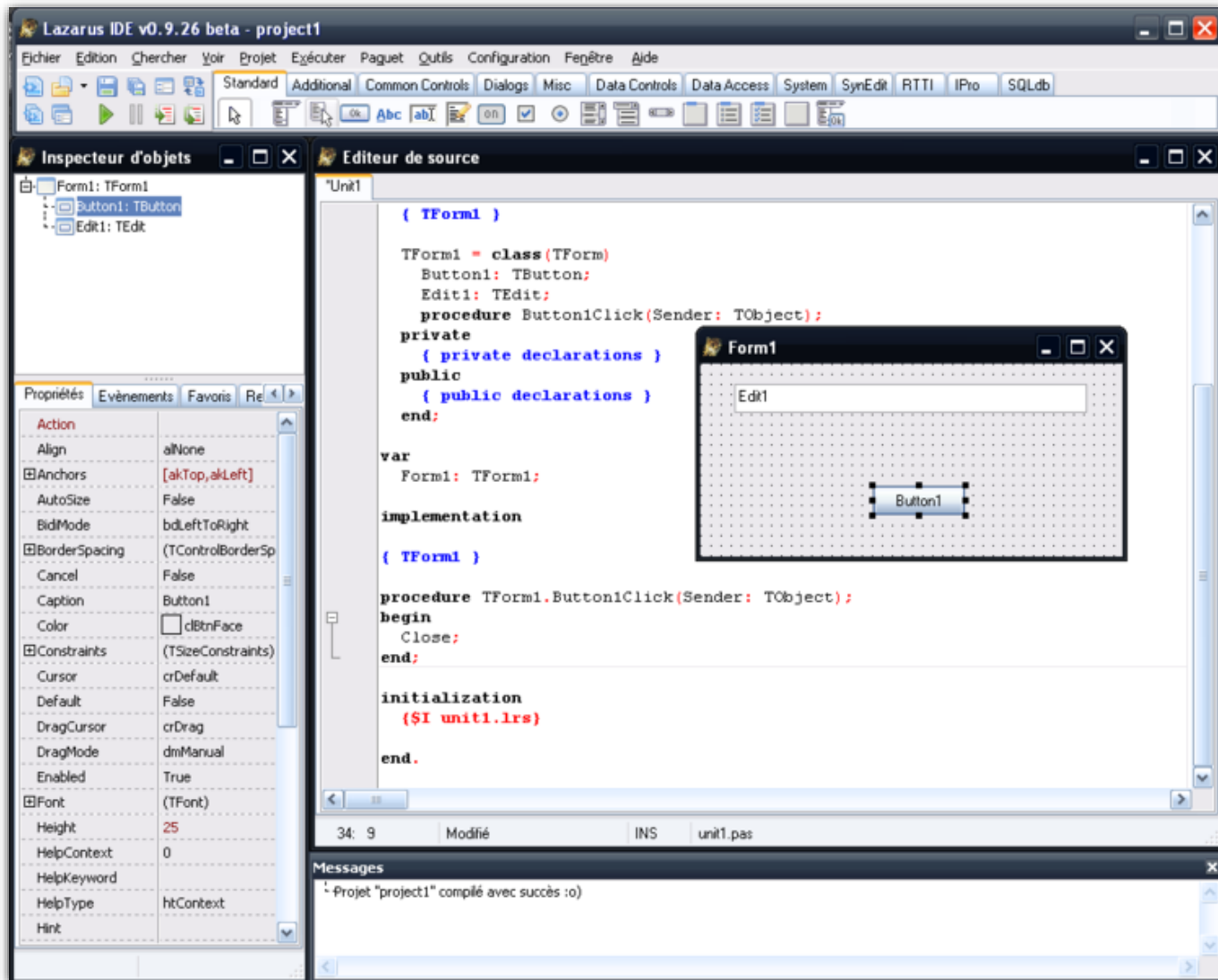
# Grundidee

## 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

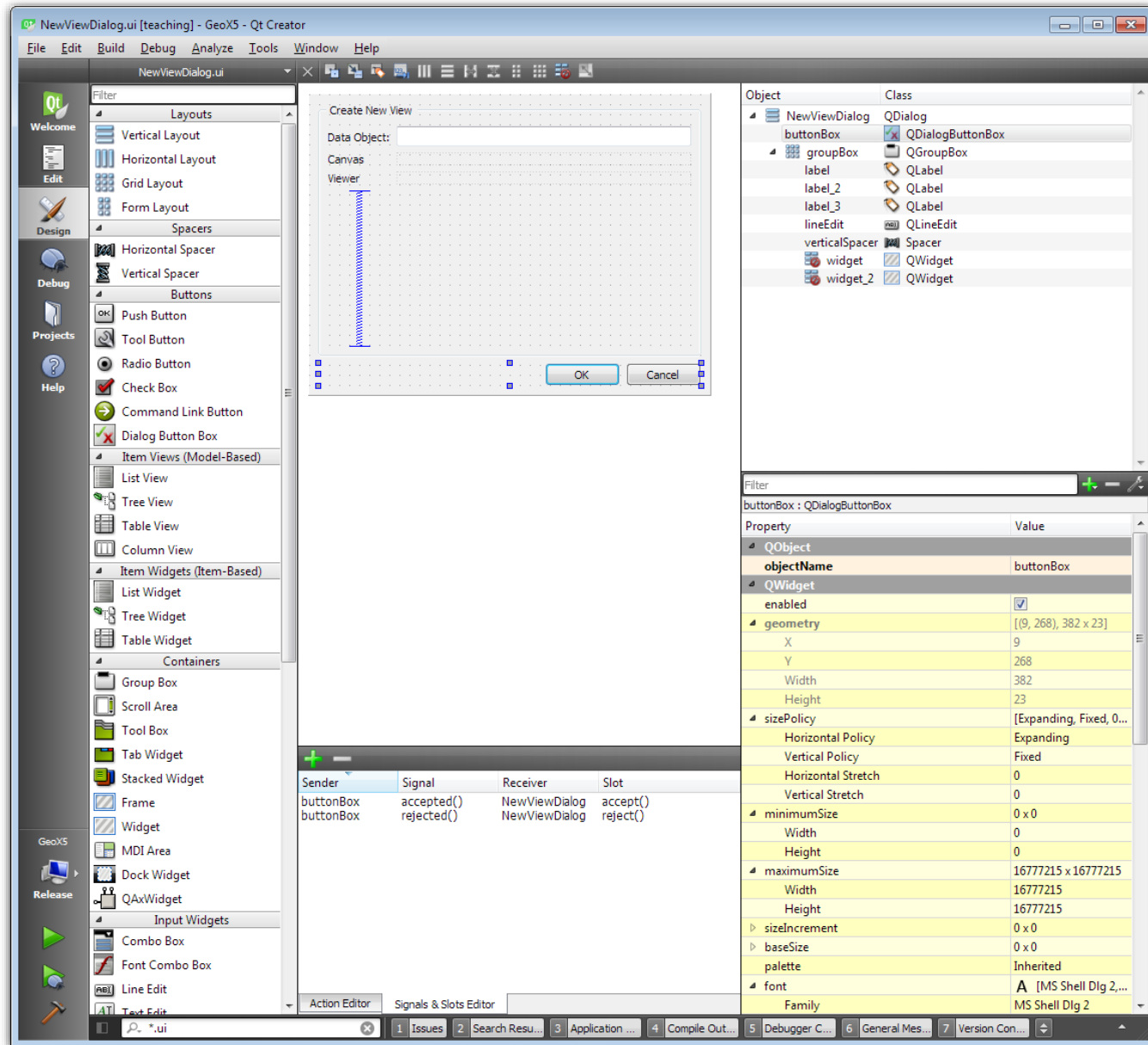


# Beispiel: Delphi / Lazarus

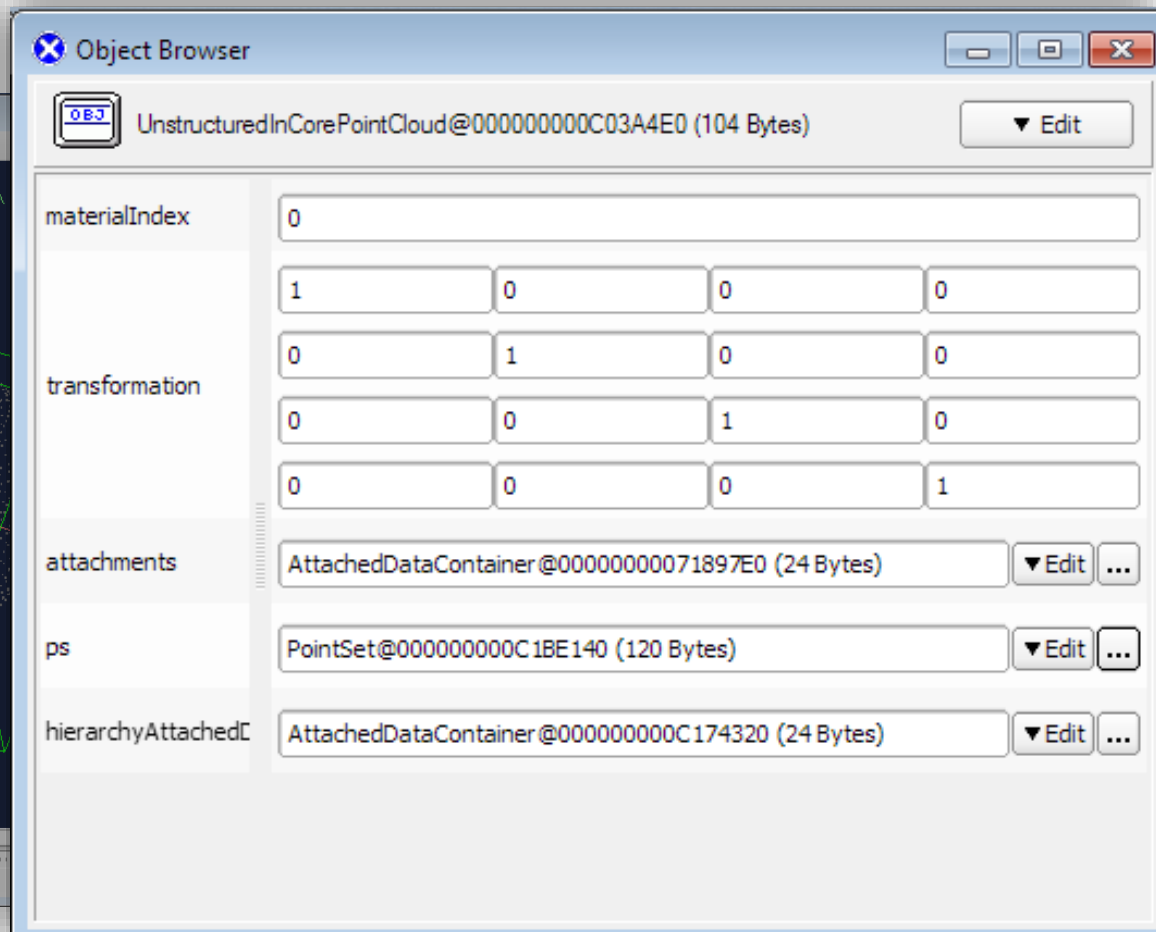
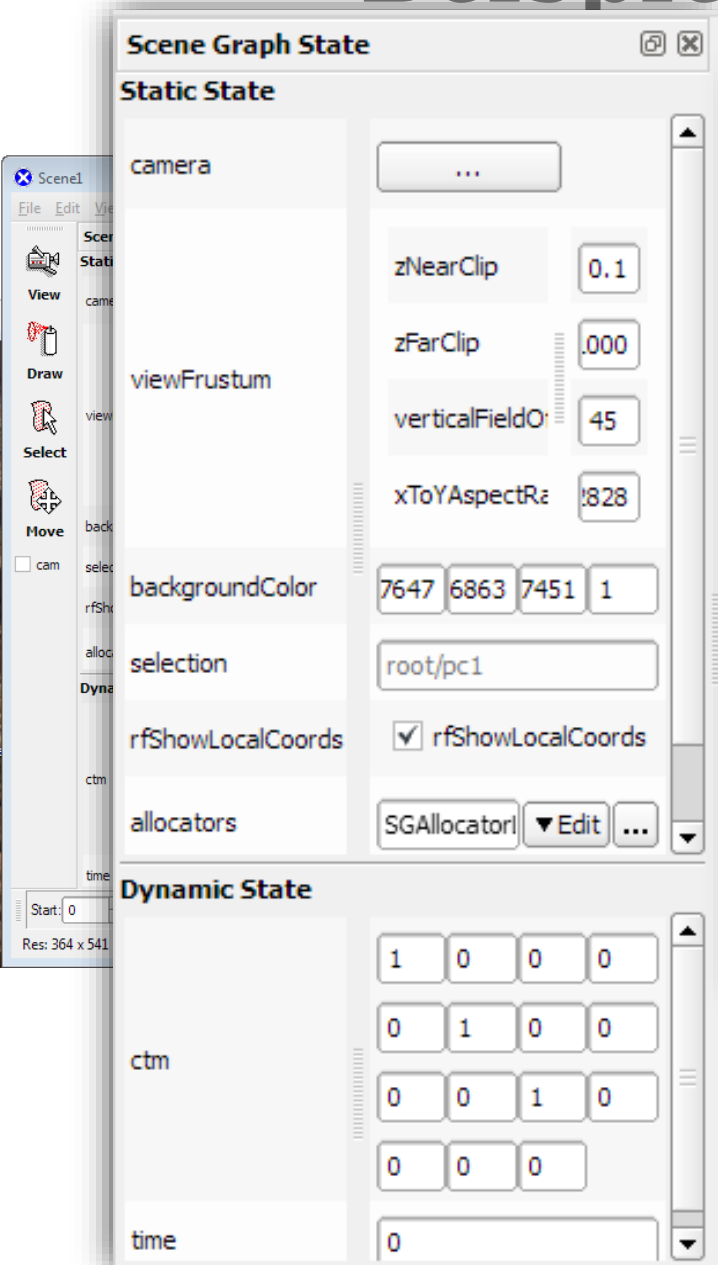


[https://commons.wikimedia.org/wiki/File:Lazarus\\_IDE\\_9-26.png](https://commons.wikimedia.org/wiki/File:Lazarus_IDE_9-26.png)

# Beispiel: QT



# Beispiele: GeoX / GeoXL



# Advanced Reflection

# Reflektion als allgemeines Prinzip

## **Reflektion/Introspection**

### **= Untersuchung von Softwareeigenschaften**

- Allgemeine Idee: Software „denkt über sich selber nach“

### **(Mindestens) zwei Prinzipien**

- Structural reflection
  - Worüber wir gerade gesprochen haben
  - Automatisieren, wenn Datenstrukturen oder Funktionen/Methoden generisch genutzt werden sollen
  - Dynamische Erweiterbarkeit
- Behavioral reflection
  - Verhalten wird an Anwendung zurückgemeldet

# Beispiele

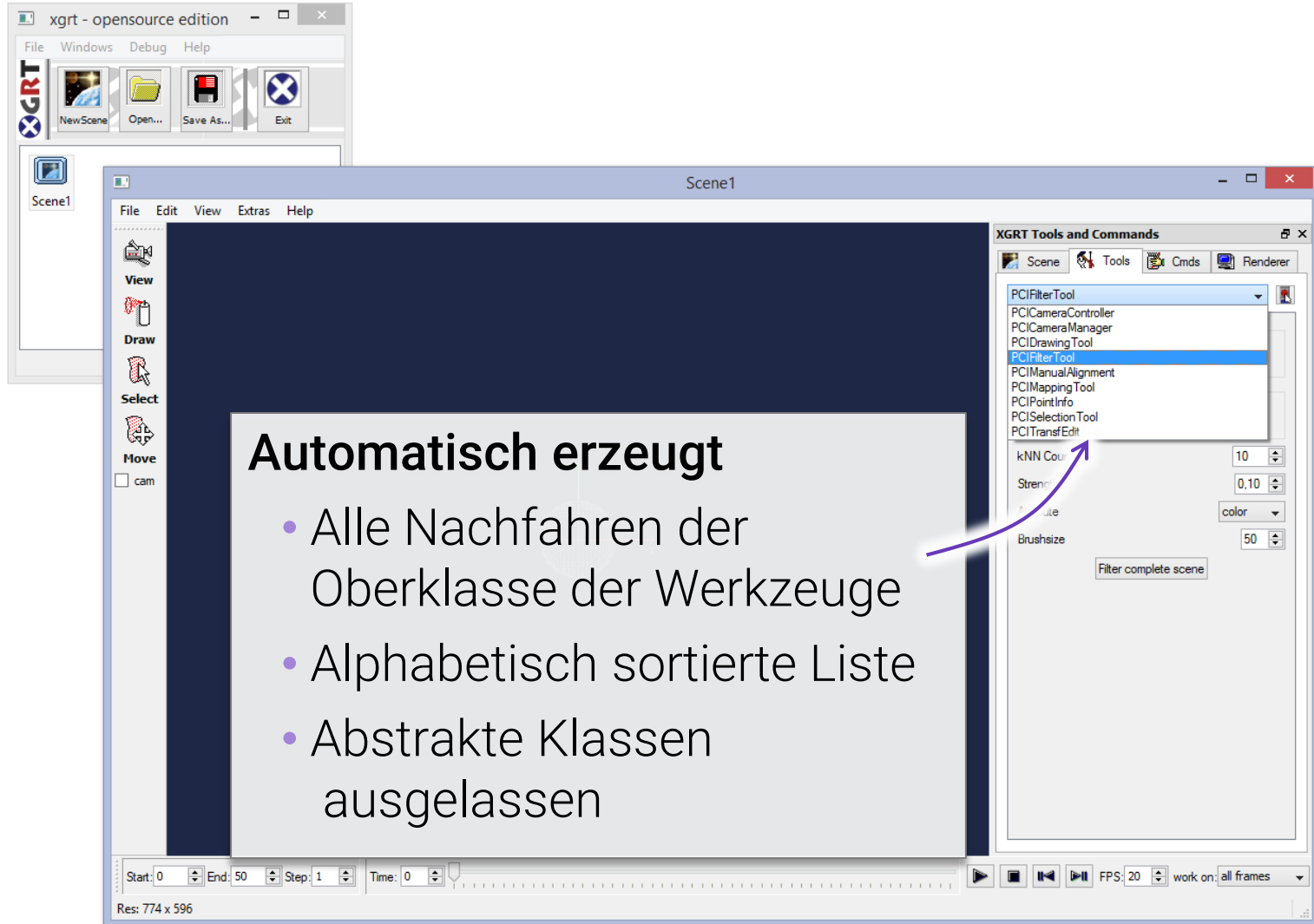
## Structural Reflection: Vererbungshierarchie

- Alle Unterklassen einer Oberklasse suchen (z.B. GeoX)
  - z.B. um alle „Shapes“ in Toolbar anzuzeigen
  - z.B. um alle Edit-Befehle in Menü anzuzeigen

## Structural Reflection: Methoden

- Menüs mit Befehlen automatisch erzeugen
  - z.B. Methoden mit Prefix „**ui\_**“ in Menü für Benutzer
  - Stärker automatisierte GUI-Programmierung
    - (Beispiel: ähnlich in GeoX für GUIs für Übungsaufgaben)
- Methoden/Ereignisse in Inspektor integrieren
  - z.B. Verdrahtung von Events in Qt oder Delphi

# Beispiel: Nachfahren erkennen



**Automatisch erzeugt**

- Alle Nachfahren der Oberklasse der Werkzeuge
- Alphabetisch sortierte Liste
- Abstrakte Klassen ausgelassen

**XGRT Tools and Commands**

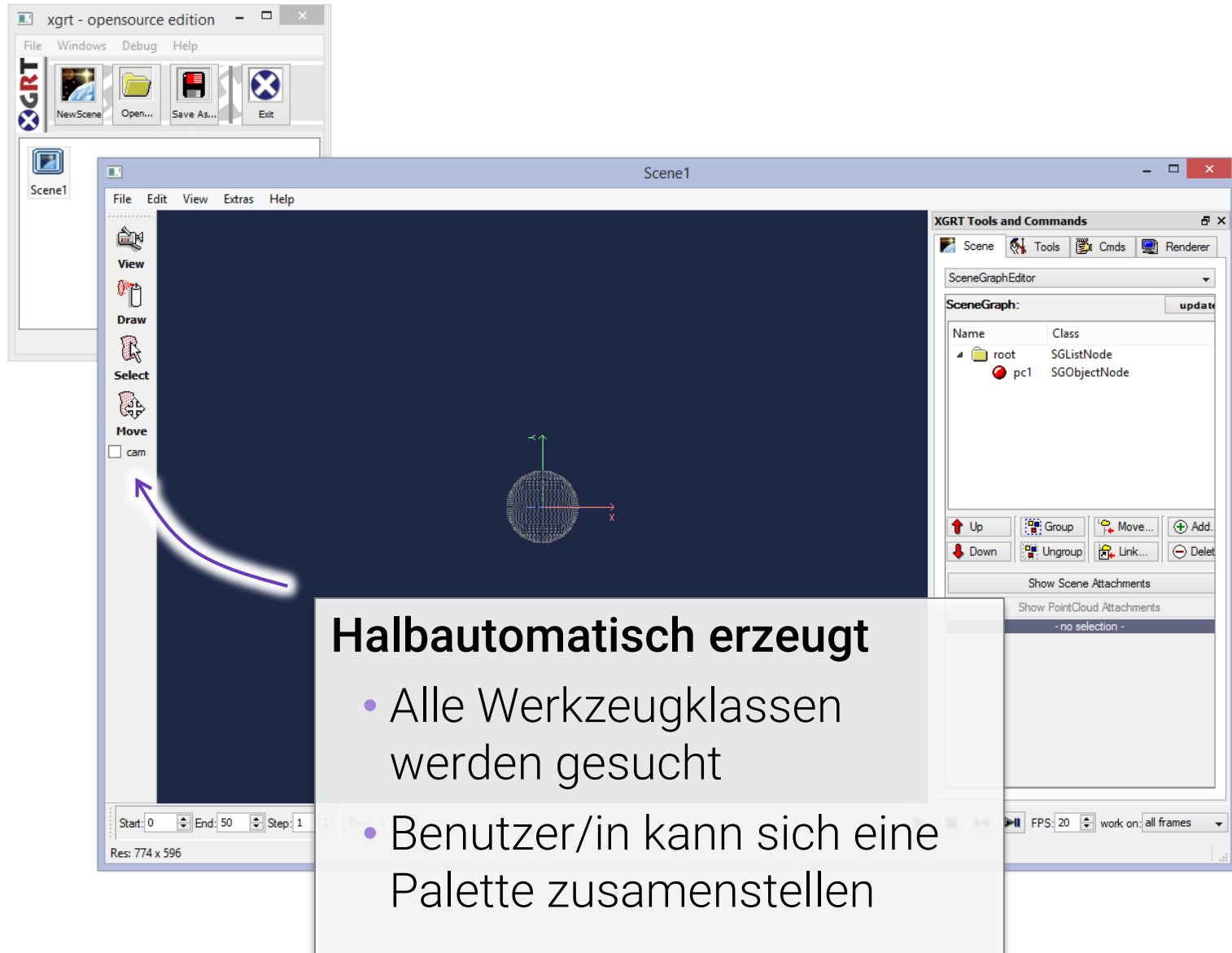
Scene Tools Cmds Renderer

PCIFilterTool  
PCICameraController  
PCICameraManager  
PCIDrawingTool  
PCIFilterTool  
PCIManualAlignment  
PCIMappingTool  
PCIPointInfo  
PCISelectionTool  
PCITransEdit

kNN Count 10  
Strength 0,10  
Brushsize 50  
Filter complete scene

Start: 0 End: 50 Step: 1 Time: 0 FPS: 20 work on: all frames  
Res: 774 x 596

# Beispiel: Nachfahren erkennen





# Beispiele

## Behavioral Reflection

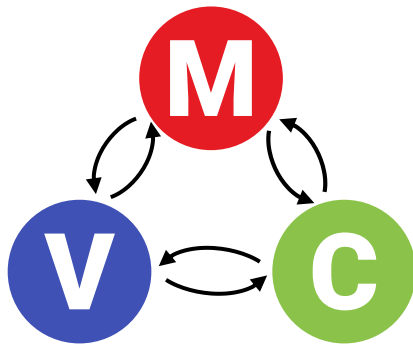
- Wichtiges Verhalten zurückmelden
  - z.B. via Ereignis-orientierter Architektur
- Verwandt mit „Aspekt-orientierter Programmierung“

## Was könnte man machen? (Ideensammlung / Anregung)

- Konstruktor / Destruktor meldet sich via Event
  - Zählen der Instanzen für Serialisierung
- Exceptions melden Events an zentrale Stelle
  - Stabilitätsprobleme erkennen und Modulen zuordnen
  - Funktionsüberwachung in verteilten Systemen
- Start und Ende von Funktionsaufrufen
  - Optimierung bei häufiger Nutzung
  - Lastverteilung in verteilten Systemen

# MVC

Model · View · Controller



Design  
Pattern

# GUI Struktur

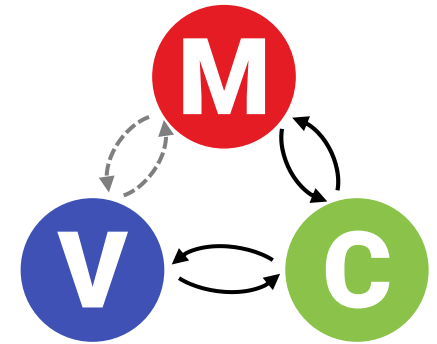
## So far...

- Widgets handle all interaction
  - „Model“: Daten speichern
  - „View“: Daten darstellen
  - „Controller“: Benutzerinteraktion (Maus/Keyboard)
- Begrenzte Wiederverwendung
  - Widgets selbst speichern die Daten
  - Zeichnen/Darstellung muss jedes Mal implementiert werden
  - Benutzerinteraktion muss jedes Mal implementiert werden
- Aufteilung erhöht Modularität

# Aufteilung: MVC

## „Model“ – Daten speichern

- Modellierung der Daten
- Unabhängig vom UI! (Anwendungsdaten selbst)



## „View“ – Daten darstellen

- Bibliothek zum Anzeigen der Daten
- Inklusive z.B.: Buttons, Rahmen, Selektion

## „Controller“ – Benutzerinteraktion

- Reagiert auf Ereignisse (Events, Maus/Tastatur)
- Aktualisiert Views, ändert Modell

# MVC & „MP“

## Meta-Pattern

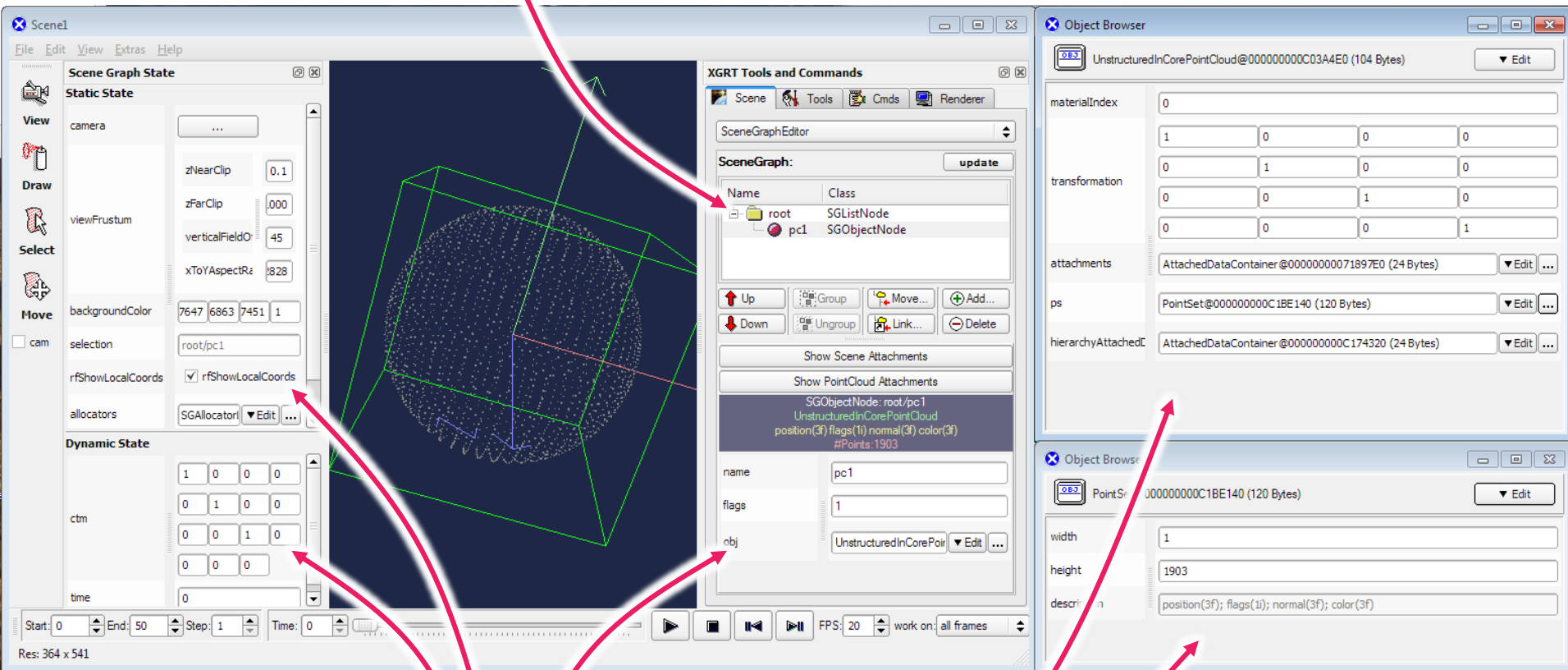
- Viele verschiedene Varianten
- Auch unter leicht unterschiedlichen Namen

## Vereinfachte Variante (sehr häufig)

- „Model-Presentation“ oder „Model-Editor“ oder „Model-[View/Controller]“
- Datenmodell separat
- Ein Editorwidget/fenster passend dazu
  - Jede Datenklasse mit einer Editorklasse assoziieren

# Beispiel für M-E: GeoX / GeoXL

Editor Klasse „Scene“



Editor Klasse „Object“ (default)

# Funktional oder Imperativ?

## MVC ist FP und OO

- Model: Reiner Datenspeicher
  - i.d.R. keine aktive Rolle
- Views: Funktionen vom Model
  - Deterministische Abbildungen
  - Seiteneffekte unerwünscht
  - Funktionales Pattern
- Controller: Kapselt State-Mutation
  - Änderung der Daten via Controller
  - Wünsche, die Daten zu Ändern werden hier umgesetzt
  - Steuerung und Modularisierung der Benutzerinteraktion

# Wie setze ich das um?

## Es gibt nicht nur ein MVC

- Viele, viele konkrete Architekturen sind „MVC“

## Beispielumsetzung

- Model
  - Kapselt die Daten
  - Abstrahiert von Speicherort und Format
    - z.B. Zugriff auf Datenbank
    - z.B. Datenstruktur im Speicher
    - Einheitliches API für Daten lesen und ändern/schreiben



# Wie setze ich das um?

## Beispielumsetzung

- View Bibliothek entwickeln
  - Eigene Komponentenbibliothek zur Visualisierung der Daten
    - Unabhängig vom Model
    - Klassen, die nützlich für Darstellung sind  
z.B. Rahmen, Labels, Edit-Boxen, o.ä.
    - Orthogonal zu Datenhaltung
  - Es kann verschiedene Domänen / Implementationen geben
    - HTML-View, PDF-View
    - OpenGL-View, Qt-View, Console-View
- View erzeugen
  - Funktionales Muster: Abbildung Model → View
    - Multiple Dispatch kann helfen: Model-Typ, Domäne

# Wie setze ich das um?

## Beispielumsetzung

- Controller
  - m.M.n. schwierigster Teil
  - Primitive für Benutzerinteraktion definieren
    - z.B. Handles, Picking, Selektion, verschieben, abstrakte Transformationen
  - Zerlegung der Benutzerinteraktion in Primitive
    - Modularisierung indem z.B. Shape-Klasse Informationen über mögliche Interaktion bereitstellt
  - Kontrolle des Views
    - z.B. Änderung der Darstellung bei Selektion
  - Komplexes Designproblem (wenig eigene Erfahrungen)
    - „Model + Editor“ viel leichter zu machen (GeoX)

# „Command-Object“ Architekturen

# Zwei Probleme

## GUIs vs. Konsole

- Leicht zu bedienen
- Schwer zu automatisieren

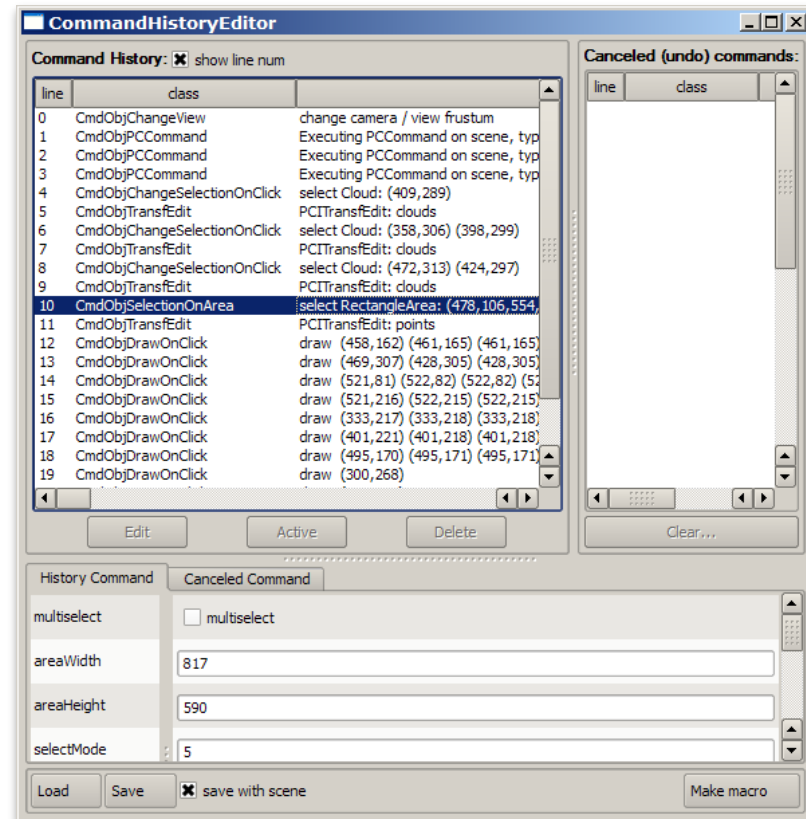
## Zwei Featurewünsche

- Undo/Redo (möglichst ohne Limits)
- Komplexes Redo (generalisierte Wiederholung)
  - Macro Recording mit Anpassungen/Generalisierung

# Command Objects

## Command Object Architecture

- Alle Aktionen werden in „*Command Objects*“ festgehalten
- Diese können *wieder abgespielt* werden *mit veränderten Parameter*
  - *Reflection* hilft hier bei der Implementation
- Literatur: [Myers et al. 96]
  - „Generalisierung“ [Myers 98]



\*) **Brad. A. Myers, David. S. Kosbi:** Reusable Hierarchical Command Objects. In: *CHI 1996*.  
**Brad A. MYERS:** Scripting graphical applications by demonstration. In: *CHI 1998*.

# Command Objects: Design

Command Object
+ parameter ( <i>zur Ausführung</i> )
+ do(root: Document) + undo(root: Document)

## Command Object Oberklasse

- Methoden für durchführung
- Mechanismus für Rückgängigmachen
  - Verschiedene Designs,
    - z.B. erzeugen ein neues CmdObj
    - z.B. Bereitstellen von undo-Methode (Änderungen speichern)
- Zentrale „Controller“-Komponente führt CmdObjs aus
  - Optionen für nebenläufige / Hintergrundberechnungen

# Nutzung für Macros

## Probleme

- Verweise auf Teile des Dokuments als Zeiger
  - Nicht auf andere Dokumente übertragbar
  - Problem bei Serialisierung (nur mit Dokument speicherbar)
- Lösung von Myers [1998]
  - Verweise grundsätzlich als Strings
  - z.B. `root.my_list[23].green_triangle.x`  
für ein Python-Objekt
  - Nun leichter transferierbar und auch editierbar
  - Nachteil: Performancenachteile
    - An dieser Stelle i.d.R. nicht kritisch

# Nutzung für Macros

## Probleme

- „Generalization“ (Verallgemeinerung)
  - Zugriff auf ein Objekt in einer Szene via `root.my_list[23].green_triangle.x`
  - Was ist Index 23?
- Ideen von Myers (...*Design your own...*)
  - Werkzeuge für interaktive Verallgemeinerung
  - z.B. „Search“-Funktion nach Objekten/Eigenschaften
  - z.B. geometrische Kriterien (wo hat die Maus hingeclickt, relativ zum aktuellen Dokument?)
  - z.B. Heuristiken (zuletzt erzeugtes Objekt gemeint, nicht Index 23)



# Allgemeines „Pattern“

## **Entwurfsmuster: „Event Sourcing“**

- Ereignisse, die Zustand des Programms ändern
- Reifizieren: Als Objekte darstellen
- Aufzeichnen / Speichern

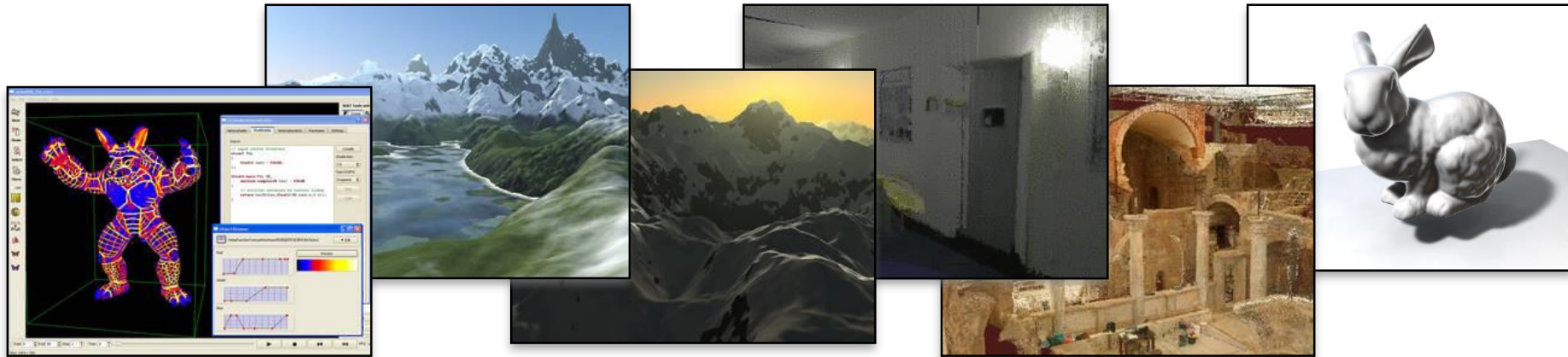
## **Wird auch in anderem Kontext benutzt**

- z.B. verteilte Systeme, die Daten speichern

# Beispiel Implementation

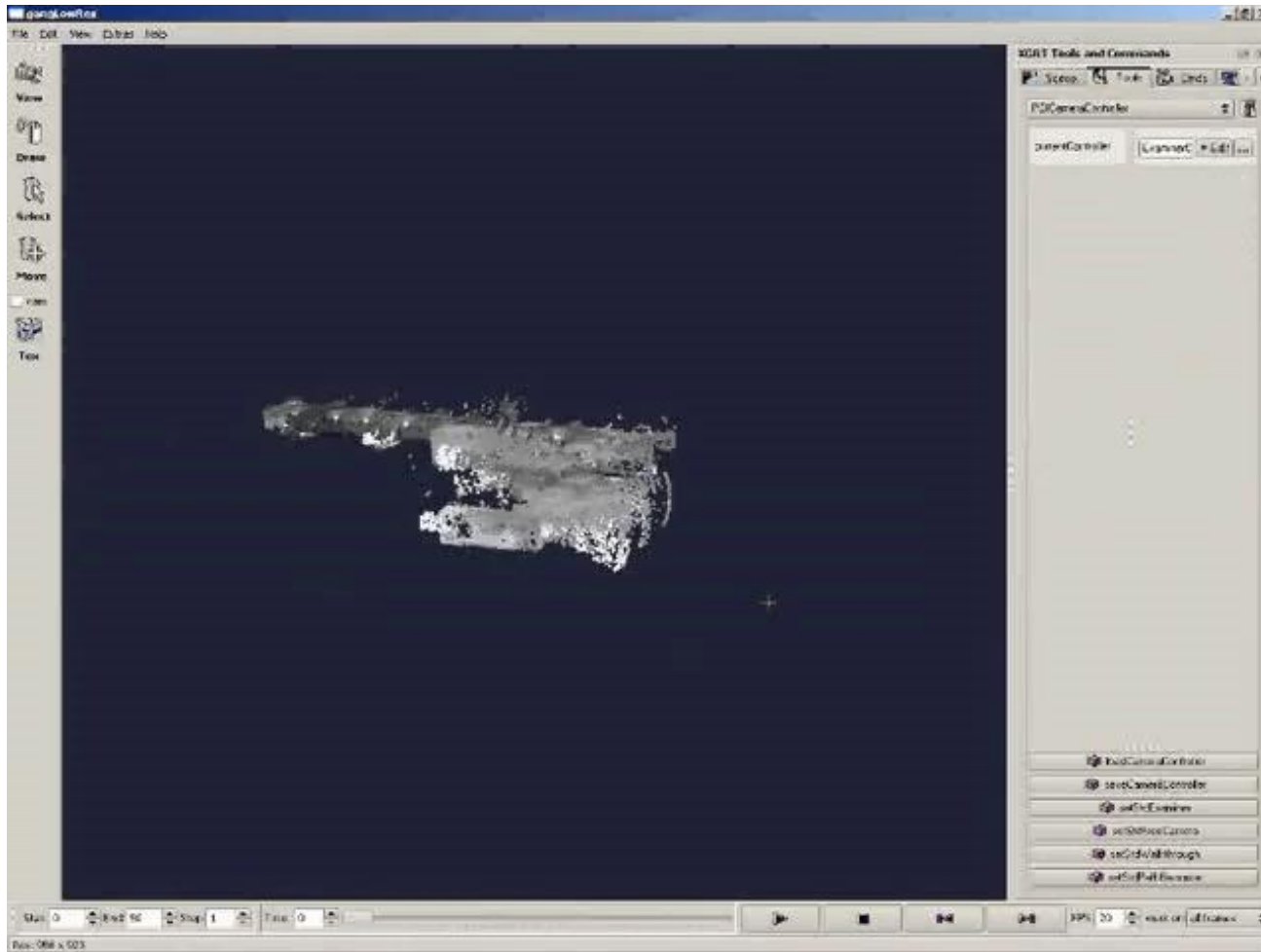
## Implementation

- Prototype editor for large point clouds
- Part of a the “XGRT” software system
  - Own work w/collaborators from 2004-2007



**M. Wand, A. Berner, M. Bokeloh, A. Fleck, M. Hoffmann, P. Jenke, B. Maier, D. Staneker, A. Schilling:**  
Interactive Editing of Large Point Clouds. *Symposium on Point-Based Graphics 2007*

# Command Scripting



**Data set:** Building Scan (76M pts/6.5 GB), P. Biber / S. Fleck, Univ. of Tübingen  
[Result from 2007: Core2 2.13Ghz, ATI X1300, 250GB/7200rpm SATA HD]

# **(F)RP –** (Functional) Reactive Programming

# Allgemein: Reaktive Programmierung

## Reactive

- War Anfang der 2010er Jahre ziemlich hip
  - Original-RFP-Paper von 1997 („*Functional Reactive Animation*“)
  - Die Idee klingt ziemlich cool

## Reactive Basics...

- Design Patterns
  - Observer
  - Iterator
  - Visitor

lösen eigentlich alle das gleiche Problem

- Der Kontrollfluss ist anders

# Patterns

## Iterator

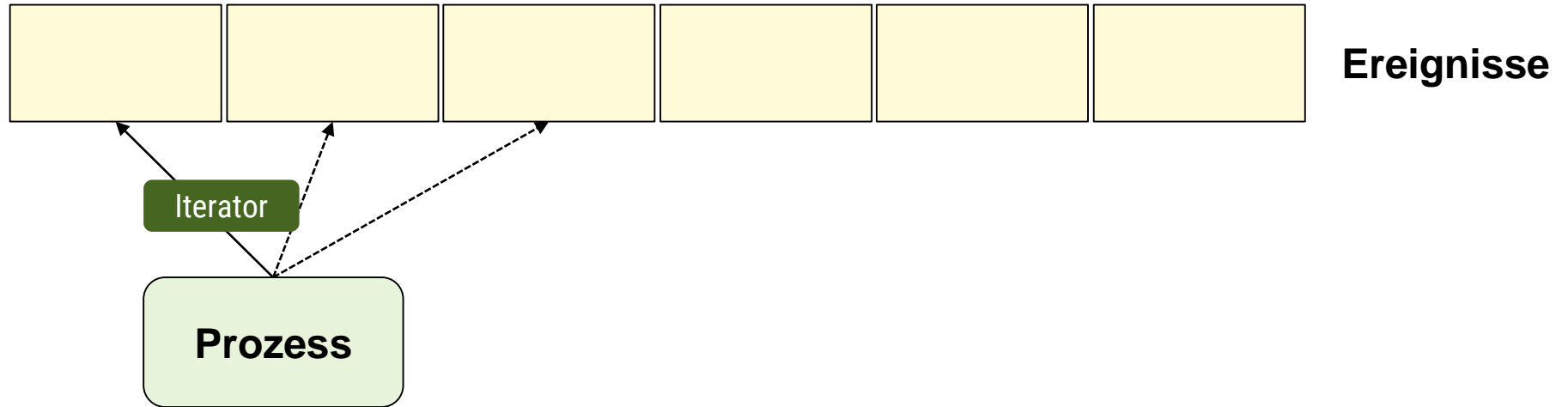
- Laufender Prozess: sucht aktiv durch Elemente
- Kontrolle über Kontrollfluss (frei programmierbar)

## Observer

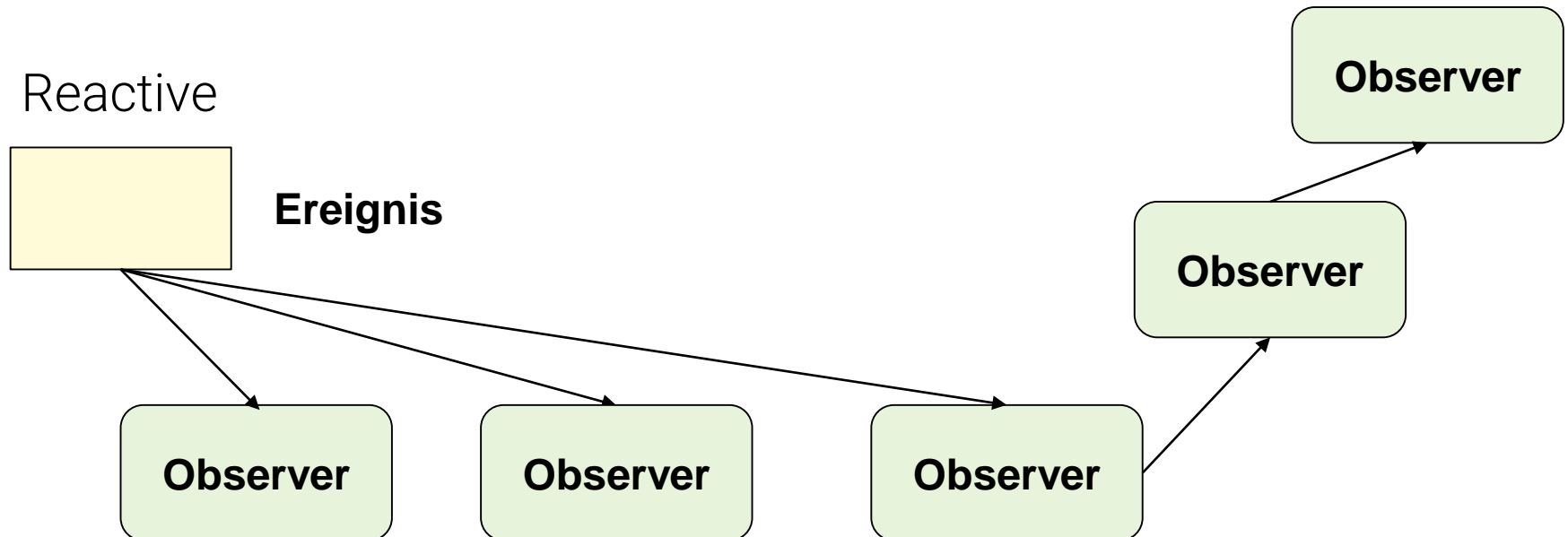
- Laufender Prozess schickt Nachrichten an registrierte Beobachter
- Möglichkeiten
  - Weiterdelegieren
  - Neue Beobachter registrieren / alte entfernen
  - Delegationsketten bauen (→ FPR)

# Iterator vs. Observer

Proactive



Reactive



Im GUI-Kontext...



# Eventhölle

## **(Weiteres) Problem mit „Widgets“ und Events**

- Spaghetti-Code
- Events können sich zyklisch auslösen
- Oft ungenügende Kapselung
- Strukturierung?
- (tbh: Events = Reactive)

# FPR

## Moderner Ansatz

- „(Functional) Reactive Programming“
- Reactive Programming ist eine längere Geschichte
  - Hier nur die Grundidee

## Datenflussgraph

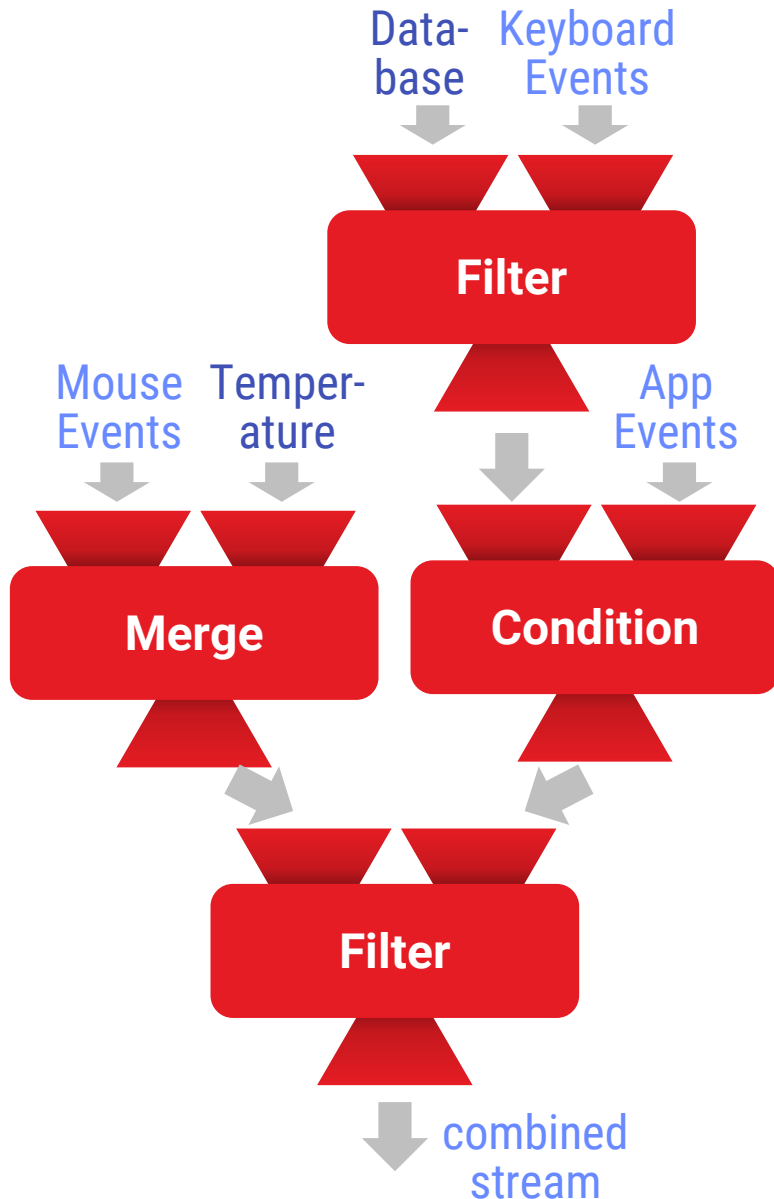
- **View** ist eine Funktion von **Model**
- Gleicher Zustand → gleicher View
- Hierarchie von Funktionen die Model in View „übersetzen“

# FPR

## Was ist mit den Events?

- **Controller** auch „funktional“ realisieren?
  - Events als Datenobjekte kapseln
  - Ströme von Events in einen Datenflussgraphen einspeisen
- Funktionale Abbildungen
  - Events (und Folgen davon) werden zu
    - Datentransformationen und
    - View-Updates

# Datenflussgraph



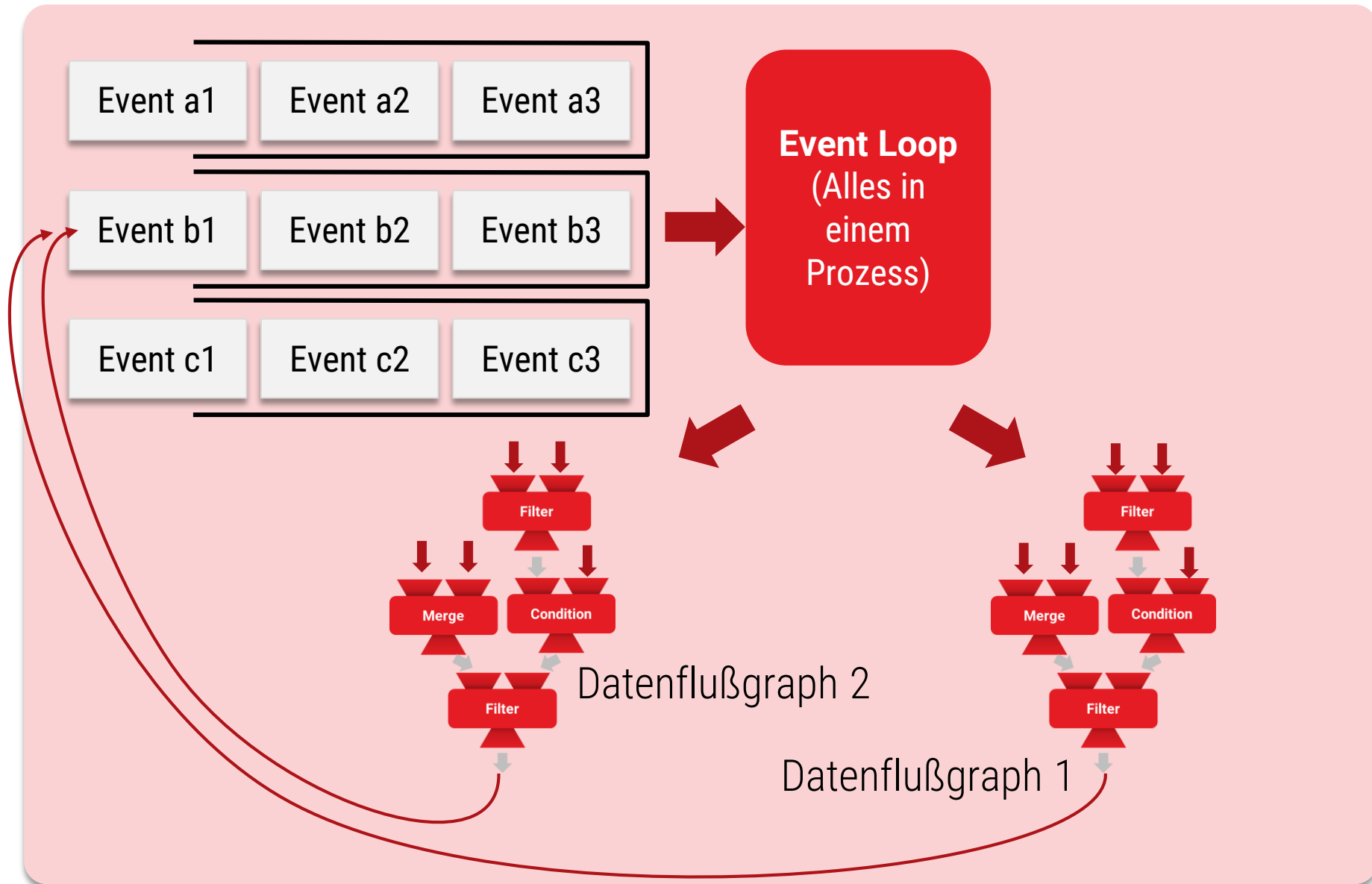
## Streams

- **Events:** Things happening
- **Values:** State of things
- Combined into streams

## Event Queue

- One event queue per stream
- Streams of changes to UI

# Implementation: Event-Loop!



# Grobe Idee

## Details?

- Nur die grobe Idee
  - Verschiedene Umsetzungen im Detail möglich
- Trade-Off zwischen
  - Oft genutzten Mustern  
(z.B. simple endliche Automaten)
  - Ausdruckstarke Interaktionsmodellierung

# Techniken für Modularisierung & erweiterbare Software

# Nützliche Techniken

## Sammlung von Techniken

- Erweiterbarkeit über symbolische Bezeichner
  - z.B. Tag-Lists, Tagged File Formats, Data-Channels, ...
  - Namens Registrierung, z.B. `com.java.awt`
- Umkehr des Kontrollflusses
  - Frage das Modul, was zu tun ist – mehr Erweiterbarkeit
- Annotation und Introspektion
  - Metadaten in Modulen (Klassen, Funktionen) bereithalten
  - Programmablauf automatisch steuern
    - z.B. Serialisierung/ Persistenz, UI-Erzeugung und -Integration
    - z.B. Online-Help und Dokumentation
  - Deklaratives Programmieren / Plug & Play: Modul laden reicht



# Speed vs. Flexibilität?

## Optimierungen

- Performance ist wichtig
  - Der meiste Code ist egal, aber kritische Teile zählen
    - Innere Schleifen
    - Intuition, was performance-kritisch sein kann
  - Profiling, dann (wenige) inner-Loops optimieren (z.B. C++, GPU)
- Techniken (jenseits von  $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n \log n)$ )
  - Caching (Vorberechnen): Aufwendige Berechnungen
  - Caching (Dynamisches):
    - Introspection, flexibler Dispatch etc. sind langsam
    - Caching auch hier möglich (z.B. Funktionszeiger vorberechnen)
    - Dynamische Code-Generierung (z.B. gcc + load\_dll).

# Polymorphie: Fortgeschrittene Methoden & Konzepte

# Polymorphie

## **Polymorphie:**

- „Verschiedene Typen via dem gleichen Bezeichner“
- Sachen, die gleich heißen, verhalten sich u.U. anders

## **Arten von Polymorphie**

- Ad-Hoc
  - z.B. überladene Methoden/Operatoren
- Subtyping
  - z.B. OOP, Interfaces etc.
- Parametrisch
  - „Generische Programmierung“, z.B. „templates“

# Polymorphie

## Polymorphie

- Ad-Hoc / Überladen von Methoden/Operatoren
  - z.B. verschiedene Parameterliste (C++/JAVA/Scala)
  - z.B. Default Parameter (Python)
  - z.B. Operatoren in allen modernen Sprachen („+“, „-“, „\*“)
- Subtyping
  - Interfaces
  - Vererbung (Traits, Mixins)
- Parametrische Polymorphie
  - Parametrisierte Typen wie `List[int]`, `List[str]`
  - Type Constraints an Parameter

# Ad-Hoc: Überladen von Methoden

## Python

- Nur default-Parameter

```
def func(x: int, y: int, twice: bool = False) -> int:  
    return x+y if twice else 2*(x+y)
```

## C++, JAVA, Scala

- Mehrfachdefinitionen, z.B. C++

```
int func(int x, int y, bool twice = false):  
    return twice ? x+y : 2*(x+y)
```

```
Shape* func(Shape* x, Shape* y, bool twice = false):  
    return twice ? new ShapeList(x,x,y,y)  
                  : new ShapeList(x,y);
```

- Unterscheidung (nur) nach Typen der Parameter

# Subtyping

## Erzeugung von Subtypen

- Vererbung (Python, Java, Scala, C++)
  - Einfachvererbung (eine „Parent Class“) in Java/Scala
  - (Beliebige) Mehrfachvererbung in Python, C++
- Interfaces
  - Rein abstrakte Klassen ohne Implementation, ohne Felder
  - Klassen können mehrere Interfaces erben
  - Interfaces können beliebig voneinander erben (azyklisch)
- Interfaces in der Praxis
  - In Java via „interface“ statt „class“
  - In Scala via traits (nächste Folie)
  - In Python/C++: Mehrfachvererbung

# Subtyping

## Erzeugung von Subtypen

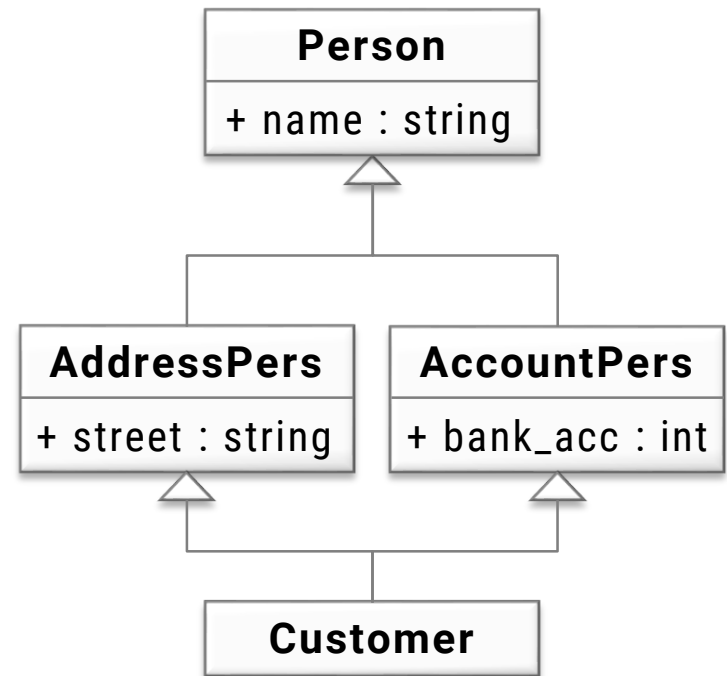
- Traits
  - Abstrakte Klassen ggf. **mit** Implementation, **ohne** Felder
  - Mehrfachvererbung wie bei Interfaces
    - Traits können voneinander beliebig erben (azyklisch)
    - Klassen können von mehreren Traits erben
  - Explizit in Scala, und modernen Java („interfaces“ mit „default methods“)
  - In Python, C++: Mehrfachvererbung

# Subtyping

## Erzeugung von Subtypen

- Uneingeschränkte Mehrfachvererbung

- Nur in C++/Python
- Probleme mit
  - Doppelten Feldern, falls mehrfach die gleiche Basisklasse geerbt wird
    - Python: Dictionary Lookup
    - C++: „virtual base classes“ zur Auswahl
  - Namensauflösung bei gleichen Namen für verschiedene Methoden
    - Prioritätsregeln nötig



*(PS: furchtbares Design)*

- Allgemeine Form nicht einfach zu nutzen



# Generische Typen

## Bereits gesehen: C++ templates

- Generische Klassen und Funktionen/Methoden

## Allgemeines Konzept

- Auch in JAVA, Scala, Python möglich
- Hier nur für Type Checks, nicht schneller
  - **C++:** anderer, optimierter Maschinencode
  - **Python:** Nur für „mypy“, kein Einfluss auf Laufzeit
  - **Scala/JAVA:** „Type Erasure“:
    - zur Laufzeit **object**-Referenzen
    - Ähnlich wie bei **mypy** nur Typprüfung (via **javac/scalac**)
    - Kein Gewinn an Laufzeitperformance

# Allgemeines Prinzip

## Komplexes Thema

- Wir besprechen die wichtigsten Konzepte
- Beispiel Python+MyPy
  - In Scala ähnlich, lediglich Interfaces für Typ-Constraints als „traits“ statt als mehrfachvererbte Klassen ausdrücken

## Allgemeines Prinzip

- Generischer Typ = Typ, der noch (mind. einen) weiteren Typ als Parameter hat
- Beispiel: `list[int]`

# Typparamter

## Definition einer generischen Funktion

```
from typing import TypeVar

# Funktionen, Methoden und Klassen können generische Typen nutzen,
# aber keine alleinstehenden Variablen
T = TypeVar('T')
def copy_list(lst: list[T]) -> list[T]:
    result: list[T] = []
    for elem in lst:
        result.append(elem.copy())
    return result
```

## Anwendung

```
x: list[int] = copy_list([1, 2, 3])
y: list[str] = copy_list(['1', '2', '3'])
```

# Definition generischer Typen

## Problem

- Wir möchten Typen für Parameter eingrenzen, z.B.

```
from typing import TypeVar
T = TypeVar('T')
def draw_list(lst: list[T]) -> None:
    for elem in lst:
        elem.draw()
```

- Wie sicherstellen, dass `T` das „`draw()`“ beherrscht?
- Optionen
  - Subtyping durch Vererbung oder Interfaces
    - „nominelles Subtyping“
  - Subtyping durch „Protokolle“
    - „strukturelles Subtyping“

# Definition generischer Typen

## Lösung Teil 1

- Wir möchten Typen für Parameter eingrenzen, z.B.

```
from typing import TypeVar
```

```
T = TypeVar('T', bound=Shape)
```

```
def draw_list(lst: list[T]) -> None:  
    for elem in lst:  
        elem.draw()
```

- Mit Type-Bounds kann man Mindestanforderungen (Oberklassen) angeben

# Definition generischer Typen

## Lösung Teil 2

- Definition des Mindesttypes

```
class Shape(object):
```

```
    ...
```

```
    def draw() -> None:
```

```
        ...
```

```
    ...
```

```
T = TypeVar('T', bound=Shape)
```

- Klasse Shape (wie in unserem Code)
  - T akzeptiert Shape und alle Nachfahren von Shape
- „Nominales Subtyping“
  - Nur dieser Typ und Nachfahren erlaubt

# Definition generischer Typen

## Lösung Teil 2

- Alternative

- Definition eines Protokolls:

```
from typing import Protocol
```

```
class Shape(Protocol):  
    def draw() -> None:  
        pass
```

```
T = TypeVar('T', bound=Shape)
```

- Nun sind alle Klassen mit einer Methode `draw()` erlaubt
- Vererbung nicht nötig!

- „Strukturelles Subtyping“

- Vorhandensein von Methoden reicht aus

# Vordefinierte Protokolle

## **Python definiert bereits nützliche Protokolle**

- Iterable, Callable, etc.
- Wird überall benutzt (z.B. for-schleifen, Signaturen für Callable Objects, etc.)



# Generische Klassen

## Typen mit Typparameter definieren

- Definition des Mindesttypes

```
T = TypeVar('T')
```

```
class NiceList(Generic[T]):  
    def __init__(self, elem1: T, elem2: T) -> None:  
        self.lst: list[T] = [elem1, elem2]
```

```
x = NiceList[int](42, 23)
```

- Klasse „NiceList“ erhält einen Typparameter
- Das gleiche Prinzip wurde bereits beim eingebauten `list[int]` angewandt

# Co-, Contra-, und Invarianz

## Frage:

- Sind `list[Circle]` und `list[Shape]` zuweisungskompatibel?

## Antwort

- Wenn ich aus der Liste lese, ist `Circle` ok, da eine Spezialisierung von `Shape`
- Wenn ich in eine Liste schreibe, ist eine Liste von `Shapes` ok wenn ich eigentlich `Circles` will (also genau andersherum)
- Fall 1 „Kovarianz“, Fall 2 „Kontravarianz“

# Co-, Contra-, und Invarianz

## Allgemeines Prinzip

- Quellen sind Kovariant
  - Auch: Rückgabetypen von geerbten Funktionen dürfen spezieller sein
- Senken / Ablageorte sind Kontravariant
  - Auch: Parameter von geerbten Funktionen dürfen allgemeiner sein
- Invarianz: Nur genau der angegebene Typ erlaubt
  - z.B. in C++ sind `vector<Shape*>` und `vector<Circle*>` völlig verschiedene Typen (Invarianz, keine Kombination von Subtyping und parametrischer Polymorphie)

# Co-, Contra-, und Invarianz

## Python?

- Varianz steuerbar:

```
T = TypeVar('T', covariant=True)
S = TypeVar('S', contravariant=True)
```

- Default ist Invarianz

## Anwendung

- Achtung, es geht um Konstrukte wie `list[T]`:
  - Ist `list[Shape]` ein Untertyp von `list[Circle]`?
- Nicht um Typebounds bez. `T`!
  - z.B. `T` muss von `Shape` abstammen, via „`bound=Shape`“

# Schlussbemerkungen

## Generische Typen

- Thema nur angerissen
- Viele Spezialitäten und Probleme im Detail
- Mechanismen und Möglichkeiten ähnlich in Scala

## Kombination Subtyping und param. Polym.

- Typprüfungsalgorithmen sind kompliziert / aufwendig
  - z.B. automatische Detektion von Ko- vs. Kontravarianz ist unentscheidbar
- Manche Sprachen verzichten daher auf Vererbung
  - „Immutability“ erleichtert auch das Problem (nur Quellen)

Abschließende Bemerkung

# Softwarearchitektur(en)

## **Viele gute (& schlechte) Ideen da draußen**

- Umschauen, Systeme studieren
- Eigenen Werkzeugkasten (weiter-) entwickeln

## **Einordnung JGU-EIS 2025**

- Motiviert durch eigene Erfahrung
  - Erfahrungen mit Implementation vor allem mit imperativen OO-Sprachen wie Delphi/C++/JAVA/Python
- Dennoch viele Grundmuster, die oft auftreten
  - Konkretes stärker durch Erfahrungen gefärbt
  - Grundideen unabhängiger von Sprache / persönlichem

# Viel Spaß beim Softwareentwickeln!

