Grundlagen wissenschaftlichen Arbeitens

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Selda Garip

MatrikelNr.: 0200012

Kennzahl: 533

seldaaa83@hotmail.com


Self
"The Power of Simplicity"

Agenda

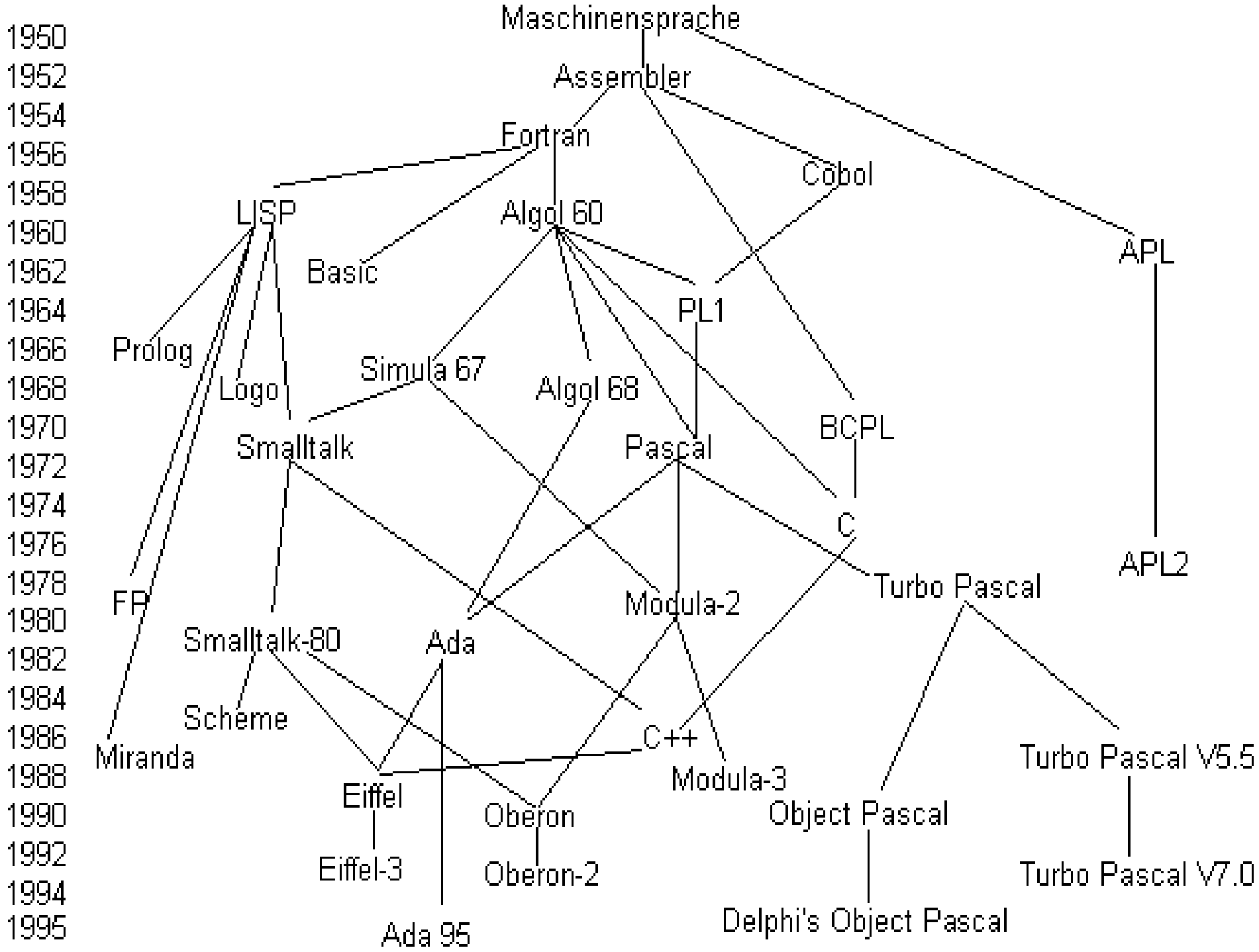
- Geschichte der OOP
- Geschichte von Self
- Definition
- Syntax und Semantik
- Besonderheiten und Anwendungen
- Vorteile & Nachteile
- Motivation über die Entwicklung

Geschichte der OOP

(kurzer Überblick)

Geschichte der OOP *(kurzer Überblick)*

- **Ursprung liegt bei der Entwicklung von *SIMULA 1987***
- **Etwa zur gleichen Zeit →Entwicklung der Programmiersprachen *ALGOL & FORTAN***
- **Populärer wird die OOP Mitte der 1980er durch → Entwicklung von *C++***
- **Ab diesem Zeitpunkt werden immer wieder objektorientierte Erweiterungen geschaffen →z.B. für *ADA,BASIC***
- **Idee → neue objektorientierte Sprachen → die sichere prozeduale Programmierung erlauben → *EIFFEL* erster erfolgreichster Versuch**


1950
1952
1954
1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1995
Maschinensprache
Assembler
Fortran
Cobol
LISP
Algol 60
APL
Basic
PL1
Prolog
Simula 67
Logo
Algol 68
BCPL
Smalltalk
Pascal
C
APL2
Turbo Pascal
FP
Modula-2
Smalltalk-80
Ada
Scheme
C++
Miranda
Turbo Pascal V5.5
Modula-3
Eiffel
Object Pascal
Oberon
Eiffel-3
Turbo Pascal V7.0
Oberon-2
Ada 95
Delphi's Object Pascal

Einige Objektorientierte Programmiersprachen

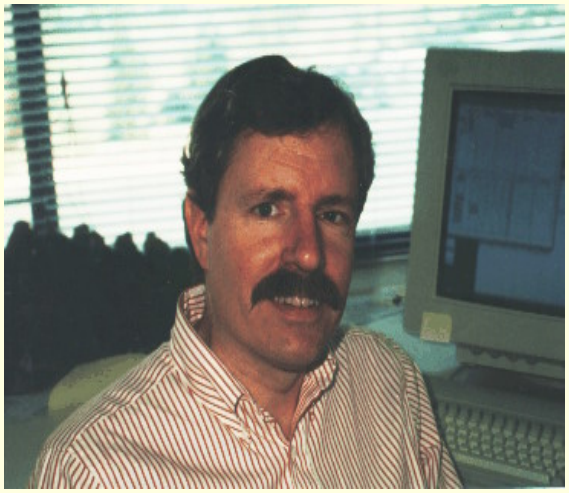
- Java
- C++
- C#
- Smalltalk
- Eiffel
- Oberon
- Object pascal, Borland
- Delphi
- Objective C
- Perl (nur eingeschränkt)
- PHP 5
- Python
- VB.NET, Visual Basic (nur eingeschränkt) ,Blitz Basic
- BlitzMax
- Visual Objects
- Modula-3
- D
- Gambas
- Modelica
- Lava
- XBase

Geschichte von Self

Geschichte von Self

- Entwicklung→ **1986** von **David Ungar & Randall Smith**
- Projekt entstand → Xerox Parc Institut → später an der Universität v. Stanford

Randall B. Smith



David Ungar

Geschichte von Self

- **1987** Veröffentlichung des ersten Self **Compilers**
- **1990** er → weitere Entwicklung bei Sun
- **1995**→ Veröffentlichung der **Version 4.0**
- **2002** → Veröffentlichung der Version 4.1.6
- **2004**→ Veröffentlichung der **Version 4.2.1**

- Grundsätzlich lässt die Self- Umgebung zum gegenwärtigen Zeitpunkt noch verwenden unter **Solaris (Sparc)** und **Mac OS X (Power PC)**

- Portierungen für andere Betriebssysteme und Architekturen sind zwar vorhanden, werden aber schon seit einigen Jahren nicht mehr aktualisiert

Definition

Was ist Self ?

Definition

- objektorientierte Programmiersprache
- prototypenbasierte Programmiersprache
- Objekt besteht aus → Slots
- Protoyp stellt ein Ablagesystem dar
- Slot stellt ein Ablagefach dar

Definition

- neue Objekte werden durch Kopie von Prototypen bzw. Objekten hergestellt

- Kopieren von Objekten → Anpassung an Bedürfnisse

„*Slots*“ Kernbestandteil von Self

Definition

- Self Objekte → eine Ansammlung von Slots
- Slot liefert einen Wert→ wenn mit einem Doppelpunkt versehen
- Slots können eine Eigenschaft repräsentieren → numerischen Wert
- Slots können eine Methode darstellen

Slotwerte können umfassen:

- einfache Datenwerte od. Wertmengen
- prozedurale Darstellung von Wissen

Definition

Existiert z.B. ein Slot namens „name „ so liefert → Konstrukt den Wert d. Eigenschaft

„name“ z.B. my Person name.

→nachfolgendes Konstrukt setzt d. Wert der Eigenschaft

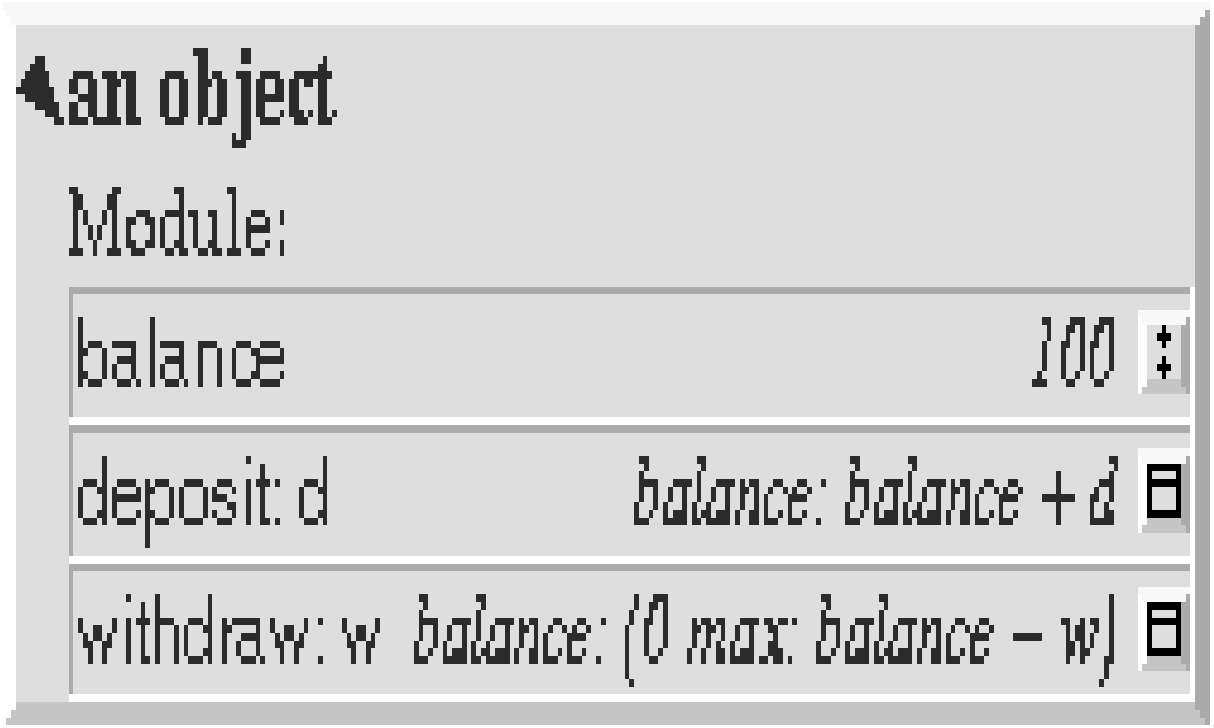
„name“ z.B. myPerson name: ´Selda`

Slots werden mit **(‘=’)** angesehen:

```
( |          parent* = traits point.
             x = 3 + 4.
             y = 5.
| )
```

Bsp.: Bankkontenobjekt

Definition



Syntax und Semantik

Syntax und Semantik

- Self → hat gleiche Syntax wie Smalltalk
- Objekt → besteht aus Slots → können Werte setzen oder zurückgeben
- anderer wichtiger Bestandteil →messages mit deren Hilfe Werte gesetzt und ausgelesen werden

Mit dem folgenden Konstrukt: **hans alter.**

wird z.B. → Name vom Objekt → **hans** geliefert um

das Alter von Hans

hingegen auf einen Wert → Beispiel **17** zu setzen

kann man folgendes eingeben**:**

hans alter: 17.

Messages

- Self arbeitet → mit messages

- Empfänger (Reciever)→ erhält eine Nachricht (Message)

'Hello, World!' print. "gibt Hello, World! aus"

3 Arten von messages:

→ **unary** (z.b., Grösse oder Länge)

→ **binary** (z.b.., + oder ~*)

→ **keywordmessages** (z.b., at:Put:)

unary message

→ besteht aus Empfänger und Nachricht, die an den Empfänger geschickt wird

→ um z.B. den sinus von 3.14 auszurechnen gibt man folgende **unary**

message ein: **3.14 sin. „schickt sin. an 3.14“**

binary- messages

→ werden verwendet um Argument zu übergeben

→ bestehen aus einem oder mehreren nicht-numerischen Zeichen

Symbol wird zwischen → Empfänger & Sender eingefügt:
z.B.

3 + 4. „sendet + mit dem Argument 4 an 3“

Keyword-Messages

→ setzen sich aus einem oder mehreren Schlüsselwörtern zusammen

folgt mit einem Doppelpunkt → dem wird Argument übergegeben

Konto anlegen: 14. „schickt anlegen mit dem Argument 14 an Konto“

→ wenn mehrere messages gemischt vorkommen

→ werden sie wie folgt abgearbeitet: → unary
→ binary
→ Keyword- messages

→ unary ; binary messages werden von → links nach rechts abgearbeitet

→ Keyword- messages werden von → rechts nach links abgearbeitet

Objekte

einfache Objekte
werden definiert mit:

(|x.y.z|)

„definiert → einfaches Objekt mit 3 Slots“

Datenobjekte

= Objekte ohne Methoden

Datenobjekte bestehen aus mehreren Slots

Objekt **( )** hat keine Slots → es ist leer

(| x = 15 y = 16 |)

hat 2 Objekte → **x** und **y**

(| a = 1. b = 2 . c. |)

→ hier werden 3 Datenslots deklariert, und zwar mit a, b, und c, die jeweils 1,2 und *nil* beinhalten

→Vertikale Striche stellen hier den Anfang und das Ende der Slotliste dar;

Methodenobjekte

- das **Methodenobjekt** enthält Code im Gegensatz zum Datenobjekt

- nachdem Auswerten des Methodenobjektes, wird nicht der Datengegenstand wieder zurückgebracht → sondern der resultierende Wert des Codes

(| a = 1. b = 2 . c. | c : self a + self b . ^ self c)

→ der Code wird hinter der Slotliste angefügt.

→ Summe von **a** und **b** wird dem Slot **c** zugewiesen;

→ sein Inhalt wird durch Operator „ **^** „ zurückgegeben

→ Punkt **.** ist ein Seperator

semantisch äquivalente Notation :

```
( | a=1. b=2. c | c: a + b. c)
```

Blöcke

- Blöcke in SELF → als *Closures* implementiert
- bestimme Steuerstrukturen werden implementiert
- Block- Literal → abgegrenzt durch eckige Klammern [ ]

Block Literal definiert 2 Objekte:

das Blockmethodenobjekt

→ beinhaltet d. Code d. Blockes

→ besitzt einen anonymen „parent slot“

das Blockdatenobjekt

→ enthält einen Elternteilzeiger und einen Slot

Operatoren

Operator besteht aus →

**@ # $ % ^ & * - + = ~ / ? < > , ; | ‘ **

„ ^ „ und **|** reserviert

Literale

- Zeichenketten: **'hello', '\n,**
- Ganzzahlen: **2, -24**
- Gleitkommazahlen: **2.1, 3.14**
- Kommentare: **„Kommentar“, „dies ist ein Kommentar“**

Ein Beispiel in SELF


```
_AddSlots: (| stack = () |).
stack _Define: (|
    stack = vector cloneSize: 100.
    topPointer <- 0.

    push: anObject = (
        topPointer: (topPointer + 1).
        stack at: topPointer Put: anObject.
    ).
    pop = (topPointer: (topPointer - 1)).
    top = (^stack at: topPointer)
|)
```

Besonderheiten & Anwendungen

Morphic

= **GUI („G**raphical **U**ser **I**nterface“)

programmieren → mit Hilfe von

Menüs, Fenstern und anderen Elementen

Welcome to SELF 4.0

If you get stuck at anytime
please look in the file help.text

To share this window with a friend on the network,
type in their machine name here.
(Your friend must be using X, and be set to allow X connections.)

share screen with: machineName:0.0

Use these buttons
to move around in
this large flat space.

Three button clicks will
move you one screenfull.

Move to the right to find out about
Mouse Button Usage, an important
first step to get you started.

Self - Konzepte in heutigen Programmiersprachen

- Virtuelle Maschine → Java
- prototypen basierte Sprache → Javascript
- Squeak → Smalltalk

Vorteile & Nachteile

Vorteile

- **Morphic (graphische Benutzeroberfläche)**

  → Komplexe Aufgaben

  → ohne viel Code

- **dynamische Laufzeitveränderung**

  →Neukompilieren entfällt

→Veränderungen am Code direkt sichtbar

Nachteile

- **keine Kapselung möglich →**

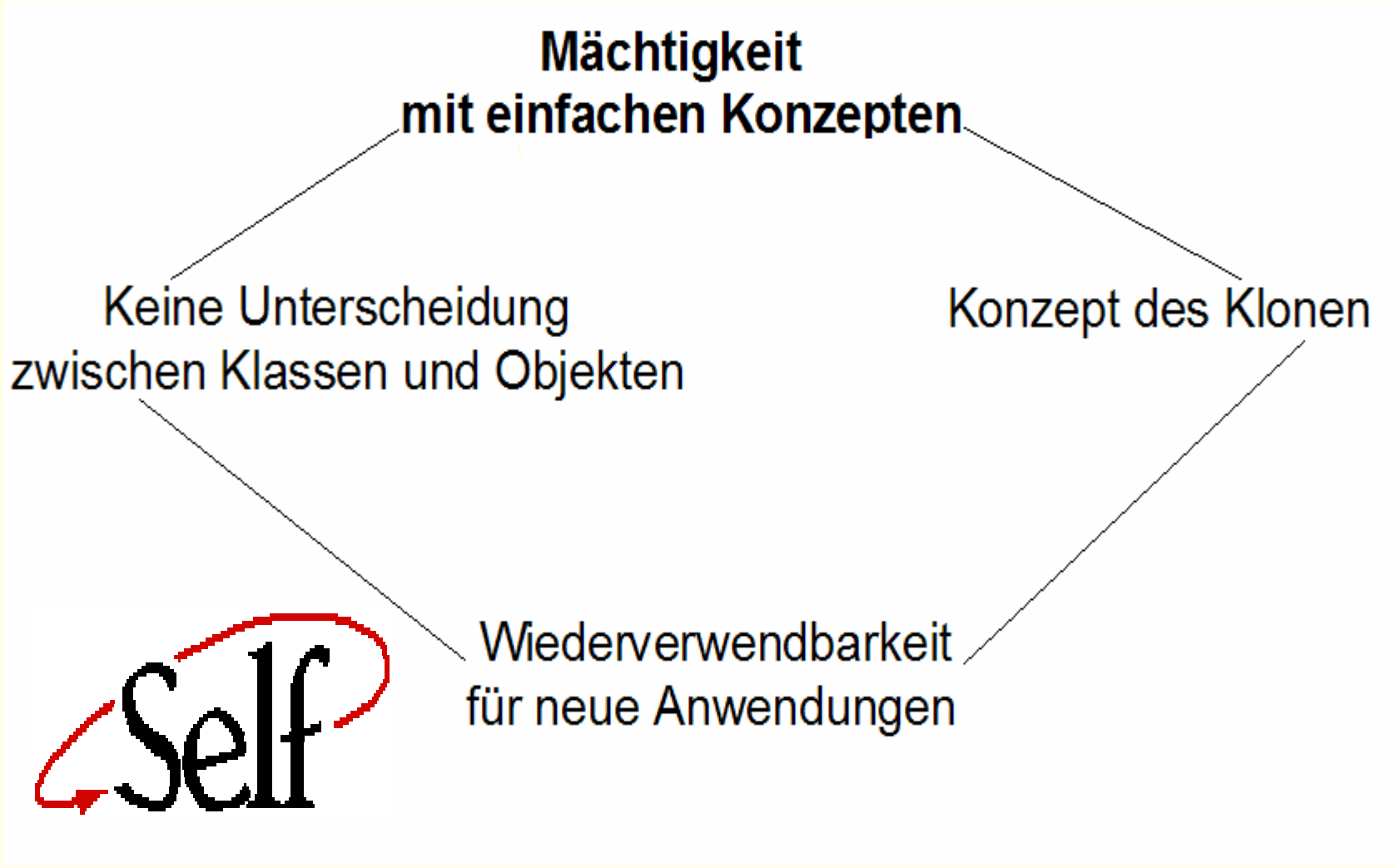
  direkter Zugriff auf Objekteigenschaften

- **Syntax ist sehr schwer zum erlernen →**

  gewöhnungsbedürfdig

Motivation zur Entwicklung

Motivation zur Entwicklung



Das „Self“ Team

- Ole Agesen
- Lars Bak
- Craig Chambers
- Bay-Wei Chang
- Robert Duvall
- Urs Hölzele
- Ole Lehrmann Madsen
- John Maloney
- Randy Smith
- David Ungar
- Mario Wolczko

Danke

für eure Aufmerksamkeit