# SELDA GARIP

Matr.Nr.: 0200012
Kennzahl:533
seldaaa83@hotmail.com



# "The Power of Simplicity"

„Grundlagen wissenschaftlichen Arbeitens"
Proseminar Programmiersprachen
Wintersemester 2005
Prof. Jens KNOOP

*„Wir müssen darauf vorbereitet sein,*
*dass wir von der Zukunft überrascht werden,*
*aber wir brauchen nicht verblüfft und sprachlos zu sein."*
(KENNETH BOULDING, aus FLYNN, 1991)


***Kurfassung**: In dieser Seminararbeit wird zunächst die Geschichte der objektorientierten Programmiersprache Self, von ihren Anfängen, ihren Paradigmen und ihrem Weltverständnis untersucht und versucht, einen Überblick über die Entwicklung und den Aufbau darzustellen. Ziel meines Berichtes ist, darzulegen, in welchem sozialen Umfeld Self (OOP) und seine Vorläufer entstanden sind und wie die typische Entwicklung bzw. Anwendung dieser Sprache verläuft*

# Inhaltverzeichnis

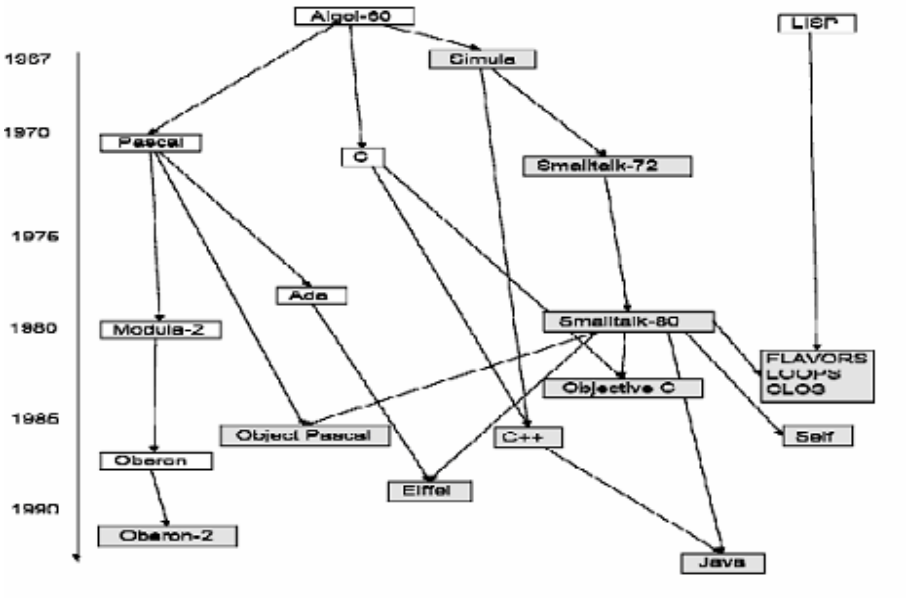
## 1. GESCHICHTE DER OBJEKTORIENTIERTEN PROGRAMMIERUNG

Der Ursprung der objektorientierten Programmierung liegt bei der Entwicklung der Sprache **SIMULA** Zur gleichen Zeit ungefähr, wurden Programmiersprachen, **ALGOL 68** und **FORTAN** entwickelt.

Populärer wurde die objektorientierte Programmierung Mitte der 1980er, hauptsächlich durch den Einfluss von **C++.** Weiter gefestigt wurde die Stellung der objektorientierten Programmierung durch die schnell wachsende Beliebtheit der grafischen Bedienoberflächen, die sich objektorientiert sehr gut programmieren lassen.

Seit dieser Zeit wurden für viele existierende Programmiersprachen objektorientierte Erweiterungen geschaffen, z.B. für **Ada**, **BASIC**, **LISP**, Pascal und andere. Erweiterungen bzw. das Hinzufügen zu Sprachen, die ursprünglich nicht für objektorientierte Programmierung entwickelt worden sind, kann zu bestimmten Problemen wie zum Beispiel der Kompatibilität und Wartbarkeit von Code führen. Im Gegensatz dazu fehlen bei rein objektorientierten Sprachen wiederum gewisse prozedurale Programmiermöglichkeiten. Um diese Lücke zu schließen, wurden verschiedene Versuche unternommen, neue objektorientierte Sprachen zu schaffen, die gleichzeitig eher sichere prozedurale Programmierung erlauben. Die Programmiersprache **Eiffel** war einer der ersten erfolgreichen Versuche, wurde inzwischen aber praktisch von **Java** verdrängt.

*Abb.1.1 Stammbaum objektorientierter Programmiersprachen*



## 2.Geschichte von Self

*David Ungar*

*Randall. B. Smith*

Self wurde mit dem Ziel entwickelt, eine Programmiersprache zu schaffen, die einfach ist und die sich stärker an der Natur orientiert als die bis dahin entwickelten Sprachen wie z.B. Smalltalk und C++. Das Projekt von **David Ungar** und **Randall B. Smith**, dass während ihrer Arbeit im Xerox-Park Institut entstand und später von Beiden an der Universität von Stanford fortgeführt wurde zeigte erste Erfolge als 1987 der erste Compiler für die Sprache entstand. Später wurde auch noch an einer Entwicklungsumgebung für die Sprache gearbeitet. In den 90er Jahren wurde die Entwicklung der Sprache schließlich bei Sun weitergeführt. Die aktuellste Version 4.2.1, die im April 2004 veröffentlicht wurde geht grundsätzlich noch auf diese Entwicklungen bei SUN im Jahr 1995 zurück als die Version 4.0 von Self veröffentlicht wurde.Grundsätzlich ist sie jeweils für die Plattform **Power PC (Mac OS X)** und **SPARC (Solaris)** als Download verfügbar. Umsetzungen für andere Betriebssysteme und Architekturen sind zwar vorhanden, werden aber schon seit einigen Jahren nicht mehr aktualisiert. [4]

### 2.1 Self- Konzepte in heutigen Programmiersprachen

Die meisten Sprachen die heute verwendet werden implementieren teilweise Ideen, die ursprünglich von Self stammen. Ein gutes Beispiel dafür ist Javascript, welches das Konzept der *prototypbasierten* Sprache übernommen hatte. Es existieren also keine Klassen sondern nur Objekte von denen die weiteren Objekte abgeleitet werden.

Auch in Java gibt es teilweise Konzepte von Self. So wird z.B. das Konzept der *Virtuellen Maschine*, die bei Self von **Urs Hölze** entwickelt wurde, auch in Java eingesetzt. Auch Smalltalk, das eigentlich als Basis für Self dient, setzte ähnliche Konzepte ein.

Die graphische Oberfläche zum Programmieren in Self, dass **Morphic** genannt wird, wurde im Projekt „**Squeak“** (graphische Oberfläche) in einer sehr ähnlichen Form auch für Smalltalk umgesetzt. [4]

# 3.Was ist Self?

Self ist eine objektorientierte Programmiersprache und wurde im Jahre 1986 von David Ungar und Randall B. Smith entwickelt. Verschiedene Quellen bezeichnen Self als einen Smalltalk Dialekt, dies ist der Sprachsyntax zurückzuführen [1] [3] [4]. Doch der Unterschied zwischen Self und dem Smalltalk liegt darin: Self ist, wie Javascript eine ***prototypenbasierte*** Programmiersprache, also ohne einer Klassendefinition, hingegen dazu, besteht Smalltalk aus Klassen und unterstützt den Zugang von Variablen, d.h. dass die wesentlichen, objektorientierte Sprachen auf Klassen basieren.
Ein Self- Prototyp besteht aus den so genannten **Slots.** Der Prototyp stellt etwas wie ein "Ablagesystem" dar, also Slots nehmen dann die rolle des "Ablagefaches" bzw. des Behälters ein. Es ist sozusagen ein Entwurfsmuster. Dieses Muster nützt prototypische Instanzen, um neue Instanzen zu erzeugen. Dazu wird eine alte Instanz kopiert und dann an die Bedürfnisse angepasst. Slots wiederum können sowohl eine Eigenschaft, wie zum Beispiel einen numerischen Wert, aber auch eine Methode repräsentieren, welche ihrerseits in der Lage ist, Nachrichten zu empfangen.
Das eigentliche Prinzip eines bestehenden Self- Objektes ist, dass es kopiert und dementsprechend abgeändert wird. Diesen Vorgang bezeichnet man auch als ***Klonen.***

## 3.1 Slots als Kernbestandteil von Self

Self-Objekte sind - wie schon erwähnt- eine Ansammlung von **"Slots".** Slots (deutsch: Schlitz) sind mit C# Properties vergleichbar. Ein Slot stellt im Prinzip einen Behälter mit dem Namen dar, die entweder Daten- oder Methodenobjekte aufnehmen können. Der Slot liefert folglich einen Wert oder setzt diesen, insofern dieser mit einem Doppelpunkt versehen ist. Existiert beispielsweise ein Slot namens **„name",** so liefert nachfolgendes Konstrukt den Wert der Eigenschaft **„name"** z.B. **myPerson name.** und das nachfolgende Konstrukt würde hingegen den Wert der Eigenschaft **„name"** setzen z.B. **myPerson name:'Selda'**

Self benutzt, wie auch Smalltalk(***typisierte objektorientierte Programmiersprache;*** entwickelt im Jahr 1970/ / **Alan Kay, Dan Ingalls, Adele Goldberg** am XEROX PARC), ***Blöcke*** *(Blöcke* werden später im Detail besprochen), um den Programmfluss zu steuern .Die Methoden entsprechen den Objekten, die Quellcode enthalten, um soeben die eigentlichen Slots zu ergänzen. Sie können innerhalb eines Self-Slots, wie jedes andere Objekt auch, eingefügt werden.
In Self wird nicht zwischen Feldern oder Methoden unterschieden. Grundsätzlich wird alles als Slot behandelt. Da die grundlegende Idee in einem Self System auf der Weitergabe von Nachrichten beruht, ist es auch nicht verwunderlich, dass vielfach eine Nachricht an „self" geschickt wird.

### 3.2 Smalltalk ⇔ Self (*kurzer Überblick)*

Smalltalk ist eine *typisierte* objektorientierte Programmiersprache und wurden in den 1970 er Jahren entwickelt.
Sie basiert stark auf den ersten objektorientierten Konzepten, die bei der Entwicklung von **Simula** gewonnen wurden, soeben wurde das erste Smalltalk-System in Simula geschrieben. Smalltalk ist im Gegensatz zu anderen Sprachen wie zum Beispiel **C++, Java** oder **C#** eine rein objektorientierte Programmiersprache, d.h. Datentypen wie zum Beispiel Integer, String oder ähnlichem sind ebenfalls vollwertige Objekte. Im Allgemeinen besteht das Smalltalk-System aus über dreihundert vordefinierten Klassen, die als Baum (Einfachvererbung) organisiert sind. Da Klassen (⇄ Self) ebenfalls Objekte sind, verfügt jede Klasse über eine implizit gegebene Metaklasse. Die Struktur eines Objektes ist durch die innerhalb der zugehörigen Klassendefinition spezifizierten und die geerbten Variablen definiert. Der Zustand eines Objektes ist durch die von ihm referenzierte Objekte bestimmt.

## 4. Syntax und Semantik

### 4.1 Allgemein

Wie schon erwähnt, hat Self praktisch die gleiche Syntax wie Smalltalk. Das Objekt in Self besteht grundsätzlich aus den sogenannten Slots, die Werte setzten oder aufrufen können.
Mit dem folgenden Konstrukt:

**hans alter.**

wird z.B. der Name vom Objekt hans geliefert. Um das Alter von Hans hingegen auf einen Wert im Beispiel 17 zu setzen kann man folgendes eingeben: **hans alter: 17**

### 4.2 Messages

Grundsätzlich arbeitet Self mit Messages. Ein Empfänger (Reciever) erhält eine Nachricht (Message). Der Empfänger interpretiert dabei die Nachricht und reagiert entsprechend mit einer Antwort darauf; wird die Nachricht von einem Punkt gefolgt, wird der Rückgabewert nicht beachtet. Ein Beispiel hierfür stellt das „Hello World“-Programm dar. [4]

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

Es werden 3 Arten von messages unterschieden → ***unary, binary und Keyword messages***

### 4.2.1 Unary messages

Bestehen 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“** |
|---|

### 4.2.2. Binary messages

Binary-Messages werden verwendet um ein Argument zu übergeben und bestehen aus einem oder mehreren nicht-numerischen Zeichen z.B. Das Symbol wird zwischen den Empfänger und dem Sender eingefügt:

| **3 + 4. „sendet + mit dem Argument 4 an 3“** |
|---|

### 4.2.3 Keyword messages

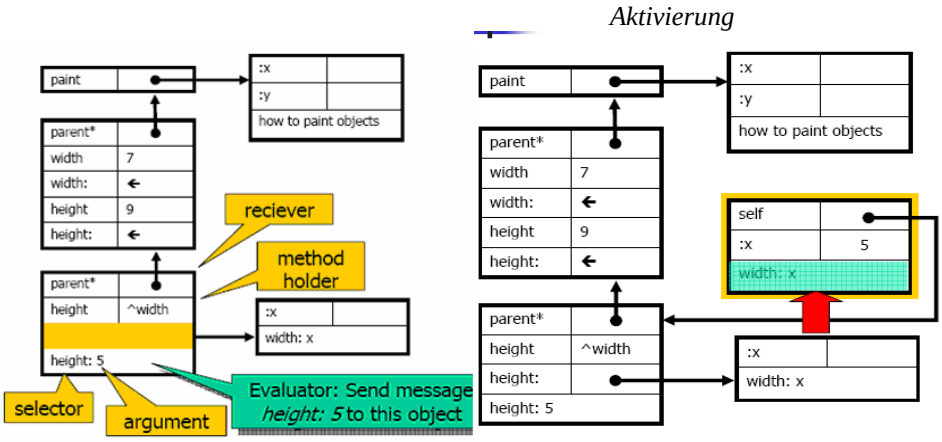
Setzten sich aus einem oder mehreren Schlüsselwörtern (Keywords) zusammen, diese werden von einem Doppelpunkt gefolgt, dem wiederum das Argument, dass übergeben wird, folgt.
Dieses kann aus Buchstaben, Zahlen oder Unterstrichen bestehen  Das erste Schlüsselwort fängt mit einem Kleinbuchstaben an, die weiteren Schlüsselwörter beginnen mit Grossbuchstaben.
z.B.:

| **konto anlegen: 14. „schickt anlegen mit dem Argument 14 an Konto“** |
|---|

| **1 min: ( 2 min : 3 Max: 4 Max: ( 5 min: 6 Max: 7 ) )** |
|---|

Wenn jetzt mehrere messages gemischt vorkommen, dann werden sie wie folgt abgearbeitet: unary -, binary **-** und dann Keyword-messages, wobei unary und binary messages von links nach rechts und Keyword- messages von rechts nach links abgearbeitet werden.

## NACHRICHTEN SEMANTIK




**Was passiert hier**

*Zunächst einmal, muss das Methodenobjekt geklont werden um die Aktivierung aufzuzeichnen. Self wird mit Slots Argumenten gefüllt. Der Code wird im Kontext der Aktivierung gewertet.*
*Parentslots sind identisch zu den unary Slots und wird wie in der Abb .mit einem Sternsuffix deklariert. Dieses angehängte Sternsymbol ist kein Bestandteil des Slotnamens, sondern eine Anweisung an SELF, diesen Slot für die interne Vererbung zu verwenden*

### 4.3 Objekte

Einfache Objekte werden folgendermaßen definiert**: (|x.y.z|)**

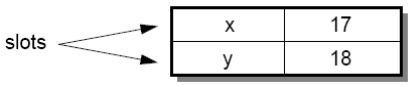
„definiert ein einfaches Objekt mit 3 Slots“ [3]

### 4.3.1 Datenobjekte

Datenobjekte sind Objekte ohne Methoden. Datenobjekte bestehen aus mehreren Slots, die durch Punkte seperiert werden. z.B.

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

Hier werden drei Datenslots deklariert, und zwar mit a, b, und c, die jeweils 1,2 und *nil* beinhalten.
Die vertikalen Striche stellen den Anfang und das Ende einer Slotliste dar; die runden Klammern deklarieren diesen Ausdruck als ein **SELF**- Objekt.
z.B. Objekt **( )** hat keine Slots → es ist leer, während das Objekt **( | x = 17 y = 18 | )** hat 2 Slots und zwar **x** und **y**




*Ein Datenobjekt kehrt, nachdem es ausgewertet ist, wieder zurück.*

#### 4.3.2 Methodenobjekte

Der Unterschied zwischen einem Methodenobjekt und einem Datenobjekt liegt darin, dass ein Methodenobjekt Code enthält und das Datenobjekt keinen. Der Code wird hinter der Slotliste angefügt. Das ganze würde dann folgendermaßen ausschauen:

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

Dieses Beispiel zeigt, dass die Summe von a und b dem Slot c zugewiesen und anschließend sein Inhalt durch den Operator „ ^ “ zurückgegeben wird. Der Punkt ist wiederum der Seperator der die einzelnen Ausdrücke voneinander trennt. Das Wort *self* führt dazu dass die Nachrichten a, b, und c, an das Objekt selbst geschickt und ausgewertet werden.

Man kann jedoch den obigen Ausdruck in einer kürzeren Schreibweise darstellen, denn in **SELF** jede Berechnung durch das Versenden von Nachrichten realisiert, wovon viel an das Objekt selber geschickt wird. Hier die semantisch äquivalente Darstellung:

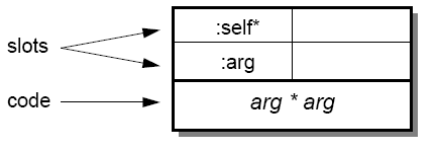
```
( | a = 1. b = 2 . c. | c : a + b . c )
```

#### 4.4 Methoden

Es gibt *ordinary methods* (oder *simply methods*), die nicht in anderen Objekten auch vorkommt. Eine Methode kann zunächst **argument slots** oder/und **local slots** haben. Eine *ordinary method* hat immer einen implizierten *parent argument slot,* der „self“ genannt wird.

**:arg**
***arg * arg***
**:self*** **(| :arg | arg * arg ):**
**slots**
**code**

Diese Methode hat ein **slot arg** → kehrt mit dem *Quadrat* seines Argumentes zurück




### ⇨ Beispiele für Methoden

*• addieren, subtrahieren, multiplizieren, dividieren, potenzieren*
*• verketten, aufteilen, ändern*
*• löschen, einfügen*
*• zeichnen, vergrößern, verkleinern*
*• starten, stoppen*
*• einstellen, entlassen• einsortieren, bearbeiten, stornieren*

## 4.5 Blöcke

Blöcke in Self sind als *Closures i*mplementiert worden, dass bedeutet, sie stellen eine Schnittstelle zum vorhandenen Kontext zur Verfügung. Ein Block- Literal *(abgegrenzt durch eckige Klammern* [ ] *),* besteht immer aus einem **Blockmethodenobjekt** *(beinhaltet den Code d.Blockes)* und einem **Blockdatenobjekt.** Das Blockdatenobjekt besitzt einen Parentslot, von dem es sein Standartverhalten erbt. und einen Slot, der das Blockmethoden Objekt enthält. Im Gegensatz zum gewöhnlichen Methodenobjekt, enthält das Blockmethodenobjekt keinen „***self Slot".*** Stattessen besitzt er einen anonymen „***parent slot".*** Anonym bedeutet, dass dieser Slot keinen Namen besitzt und somit auch nicht explizit angesprochen werden kann.

Um einen Block zu aktivieren, wird ihm eine Nachricht, namens *value* geschickt. Wenn zusätzlich noch ein Argument übergeben werden soll, kann das durch eine Nachricht- *value* – geschehen. Bei mehreren Argumenten werden diese durch *With* angegeben.

Ein Beispiel zu einem aktivierten Block mit drei Argumenten:

```
[ | :arg1 . :arg2 . arg3 | ...] value : 1 With : 2 With : 3
```

### 4.6 Operatoren

Operatoren bestehen aus einem oder mehreren der folgenden Zeichen:

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

Zwei Zeichen sind reserviert und sind nicht Operatoren ⇨

```
|          ^
```

Diese 2 reservierten Zeichen müssen in Kombination mit anderen Operatorsymbolen auftreten z.B.:

```
|| , |> , ^+ , ^^, ^|
```

Produktion:

*op-char* → ‘!’ | ‘@’ | ‘#’ | ‘$’ | ‘%’ | ‘^’ | ‘&’ | ‘*’ | ‘-’ | ‘+’ | ‘=’ | ‘~’ | ‘/’ | ‘?’ | ‘<’ | ‘>’ | ‘,’ | ‘;’ | ‘|’ | ‘‘’ | ‘\’

*operator* → *op-char* {*op-char*}

### 4.7 Literale

Beispiele für die wichtigsten Literale in Self:

Zeichenketten: **'hello', '\n'**

Ganzzahlen: **2, -24**

Gleitkommazahlen: **2.1, 3.14**

Kommentare: „**Kommentar“, „dies ist ein Kommentar“**

## 5. Besonderheiten von Self

### 5.1 Morphic

Eine der Ziele von Self war eine einfache Bedienung der Sprache zu ermöglichen.
Zu diesem Zweck wurde eine graphische Benutzeroberfläche namens **Morphic** geschaffen, mit der man mit der Hilfe von Menüs, Fenstern und anderen Elementen programmieren kann. Grundsätzlich startet der Benutzer auf einer Arbeitsfläche auf der er Objekte einfügen und verändern kann. Diese Objekte bestehen wiederum aus Slots, die Daten oder Methoden aufnehmen. Mit Hilfe von Linien kann man die Vererbung und Abhängigkeit von Objekten verändern.

### 5.2 Veränderung während der Laufzeit

Der Code eines Programms in Self lässt sich während der Laufzeit verändern. Man muss also nicht wie in C oder Java nachdem man den Code verändert hat neu kompilieren und das Programm neu starten. Die Änderungen können während das Programm läuft durchgeführt werden und wirken sich direkt auf das Verhalten des Programms aus.

## 6.Vorteile

- Einheitlichkeit- alles ist ein Objekt
- Einzelne Module können leichter wieder verwendet werden
- Komplexere Aufgaben ohne viel Code zu schreiben → durch Programmierumgebung **Morphic**
- Der Code wird leichter erweiterbar
- durch dynamische Laufzeitveränderung erzielt man schneller ein fertiges Programm
- ein Objekt hat eine nach außen definierte Schnittstelle, mit der andere Objekte kommunizieren können → bessere Kommunizierbarkeit

## 7.Nachteile

- keine Kapselung möglich, es kann also von jedem Teil des Programms direkt auf Objekteigenschaften zugegriffen werden
- keine Typprüfung
- Syntax von Self ist eher gewöhnungsbedürftig, also sehr schwer zum erlernen, besonders für Leute, die Java oder C++ erlernt haben

## 8.Motivation zur Entwicklung der Sprache

Nach ***Smith*** und ***Ungar*** hatten objektorientierte Sprachen eine Schwäche, nämlich die Unterscheidung zwischen Klassen und Objekten. In Programmiersprachen wie Java, C++ oder auch Smalltalk werden als erstes Klassen geschaffen. Diese stellen eine Schablone oder auch Bauplan für ein Objekt zur Verfügung. Ein neues Objekt muss jetzt instantiiert werden, d.h. es wird ein neues Objekt nach der Beschreibung, die in der Klasse steht, erzeugt.
In Self wurde auf diese Unterscheidung verzichtet. Grundsätzlich gibt es nur Objekte, die nicht auf Klassen angewiesen sind. Eine Instanz stellt im Fall der objektorientierten Sprache eine sog. Klasse dar. Um ein neue Instanz zu schaffen wird ein bereits vorhandenes Objekt einfach kopiert bzw. geklont.
Eines der hauptsächlichen Gründe für die Entwicklung von Self war, eine mächtige Sprache mit einfachen Konzepten zu schaffen [1].

## 9.Zusammenfassung

*Self wurde von David Ungar und Randall B. Smith entwickelt um eine einfachere und doch mächtigere Sprache als die bis dahin entwickelten OOP-Dialekte zu schaffen. Sie ähnelt mit der Syntax des von Smalltalk und wird deshalb häufig auch als Smalltalk-Dialekt bezeichnet. Self unterscheidet sich allerdings in einigen wesentlichen Punkten, von Smalltalk. Self ist eine Prototypen basierte Sprache: verzichtet also gänzlich auf Klassen. Objekte werden nicht instanziert, sondern man erstellt direkt eine Kopie des bestehenden Objektes und verändert dieses.*

## 10.Quellenverzeichnis

### Literaturliste


[1] **Self: The Power of Simplicity**/ *David Unger and Randall B. Smith* 1987

[2] P**rogramming as an Experience: The Inspiration for Self** / *Unger and Smith*

[3] **Prototype-Based Application Construction Using SELF 4.0 /** *Mario Wolczko,R. Smith*


### Websites


[4] Self
http://de.wikipedia.org/wiki/Objektorientierte_Programmierung

[5] http://www.self-support.com/

[6] http://informatik.techniktoday.de/Self_(Programmiersprache)

[7] http://research.sun.com/self/

[8] http://research.sun.com/self/release_4.0/Self-4.0/Tutorial/ index.html


### The Movie


[9] **“Self” /** *Randall B. Smith´s Introduction*