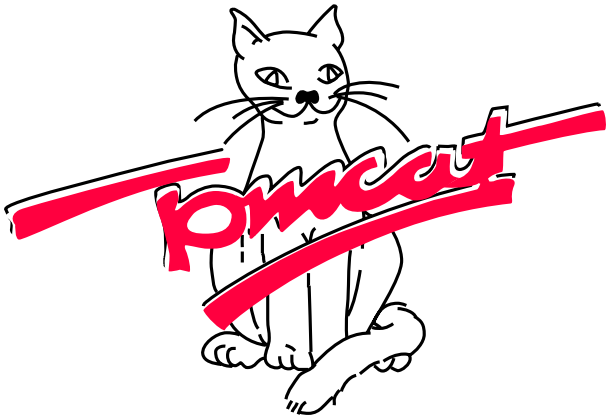
Objektorientierte Software-Entwicklung mit Smalltalk

Michael Prasse



Tomcat Computer GmbH

Inhalt

1. Geschichte von Smalltalk
2. Einführung in das objektorientierte Paradigma
3. Programmieren mit Smalltalk
4. Entwickeln graphischer Benutzeroberflächen
5. Sprachtheoretische Betrachtung von Smalltalk

Beispiel-Anwendung

Entwickeln einer einfachen Adressverwaltung

- Programmlogik
- modularer Aufbau
- MVC-Architektur

Thementafel

Nr.	Themen
1	Einführung und Entwicklungsgeschichte von Smalltalk
2	Objektorientierung und Philosophie von Smalltalk
3	Smalltalk - Programmieren im Kleinen
4	Einführung in die Visualworks-Entwicklungsumgebung
5	Wichtige Klassen
6	Überblick über die Klassenbibliothek
7	MVC-Architektur - Standard MVC - ApplicationModel - Subcanvas
8	Smalltalk-Idiome
9	Sprachtheorie - Reflexion - Metaklassen - Typbetrachtung

Literatur und Ressourcen

Objektorientierung

- Link-Sammlung Cetus, www.cetus-links.org
- Object FAQ, iamwww.unibe.ch/~scg/OOinfo/FAQ/index.html
- Meyer, B.: „Object-oriented Software Construction“. Prentice Hall. 1997.
- UML, www.omg.org; www.rational.com

Smalltalk

- Goldberg, A.; Robson, D.: „Smalltalk-80: The Language“. Addison Wesley. 1989.
- Kent, B.: „Smalltalk Best Practice Pattern“. Prentice Hall. 1997.
- Skublics, S.; Klimas, E.J.; Thomas, D.A.: „Smalltalk with Style“. Prentice Hall. 1996.
- Lewis, S.: „The Art and Science of Smalltalk“. Prentice Hall. 1995.
- Lalonde, W. R; Pugh, J. R.: „Inside Smalltalk“. Prentice Hall. 1990.
- Bücker, M.C.; Geidel, J.; Lachmann F.: „Programmieren in Smalltalk mit Visual-Works“. Springer. 1995.
- Mittendorfer, J.: „Objektorientierte Programmierung mit C++ und Smalltalk“. Addi-son-Wesley. 1990.
- Howard, T.: „The Smalltalk Developer's Guide to VisualWorks“ SIGS. New York. 1995.
- Krasner, G. E.; Pope, S. T.: „A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80“. In: „Journal of Object-Oriented Programming“. Heft 3. 1988. S. 26-49.

- Kay, A.: „The Early History Of Smalltalk“. In: „ACM Sigplan Notices“. Band 28. Heft 3. März 1993. S. 69-94.
- Goldberg, A.: „The Community of Smalltalk“. In: Salus, P. H.: „Handbook of Programming Languages“. Band 1. Macmillan Techn. Publ. 1998. S. 51-94.
- Tomek, I.: „The Joy of Smalltalk: An Introduction to Smalltalk“. 2000. wiki.cs.uiuc.edu/VisualWorks/Joy+of+Smalltalk
- Smith, D.: Smalltalk-FAQ, www.dnsmith.com/SmallFAQ/
- Malik, V.: Smalltalk-FAQ, www.ipass.net/~vmalik/
- Painter, A.; Turner, A. J.: „Introduction to Visualworks: An Application Approach“ 1995. www.cs.clemson.edu
- Johnson R.: „Object-Oriented Programming and Design with Smalltalk“. 1996. st-www.cs.uiuc.edu/users/johnson
- Cincom-Tutorials www.cincom.com/scripts/smalltalk.exe/education/index.asp?content=tutorials

1. Entwicklung von Smalltalk

1.1. Ausgangspunkt um 1965

- Mainframes
- keine graphischen Terminals, keine hochauflösenden Monitore
- keine Personalcomputer
- Batch-Betrieb

1.2. Einflüsse

„Sketchpad: A man-machine graphical communication system“ (Ivan Sutherland 1963)

- interactive computer graphics
- clipping and zooming windows
- Ausdrücken der Anwendungslogik durch Constraints

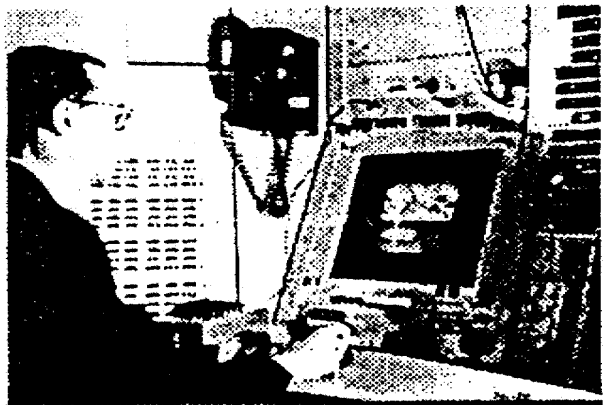
Simula (Dahl, Nygaard 1966)

- objektorientierte Grundzüge (Klasse, Objekte)
- Sprache für Simulation (Co-Routinen)
- Basiert auf Algol

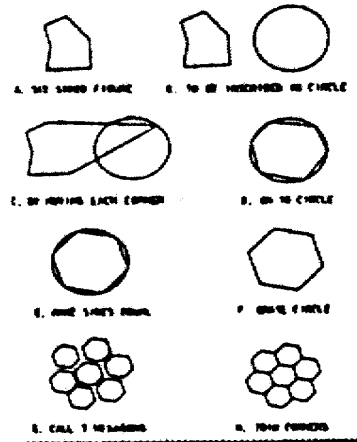
Grundlegende Ideen:

- „master descriptions that could create instances, each of which was an independent entity controlled either by a system of constraints or by a procedural programming language“ (Goldberg 1997)

- „The basic principle of recursive design is to make the parts have the same power as the whole“ (Bob Barton)

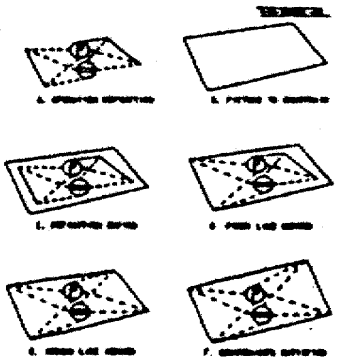


When there was only one personal computer
Ivan at the TX-2 ca. 1962



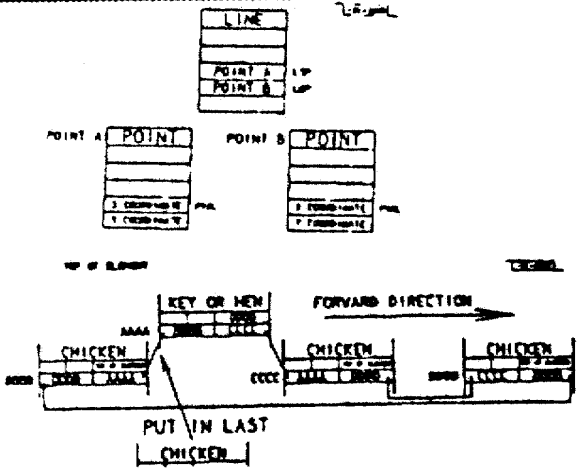
Drawing in Sketchpad

Constraints represented as icons

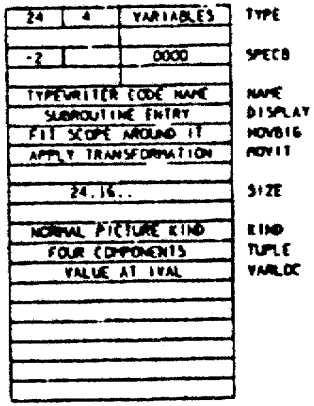


Constraints merged with picture

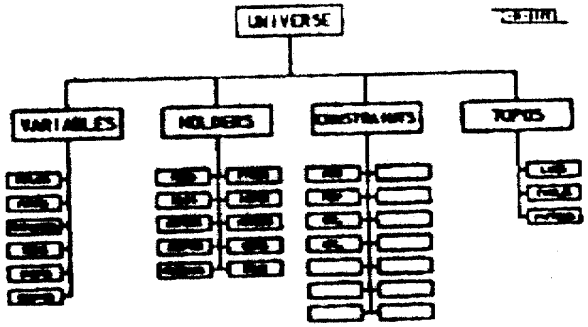
Programming with constraints



Sketchpad Structures



"generic block" showing procedural attachment



Sketchpad's "inheritance" hierarchy

(aus Kay 93)

Logo
- Programmiersprache für Kinder
- flexibel, erweiterbar
- benutzerfreundlich, einfach

Maus (Stanford Research Laboratory 1965, Doug Engelbart)
- Ersetzen des Lichtstiftes
- praktische Erprobung und Weiterentwicklung im Xerox PARC ab 1970.
- erste kommerzielle Nutzung als Teil des Xerox Star (1981)

Die Entwicklung von Smalltalk ist eng verknüpft mit der Entwicklung von:
- „personal computing“
- Human Computer Interaction (HCI)
- personal workstations

1.3. FLEX Machine (Alan Kay 1968)

'Flex' - 'Flexible Extensible Language'
'Flex Machine' - 'reactive engine'

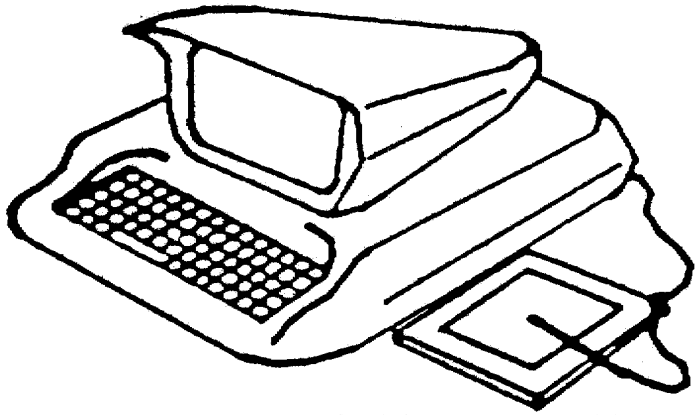
➡ Ausgangspunkt der Smalltalk-Dialekte

- Idee des „personal computing“

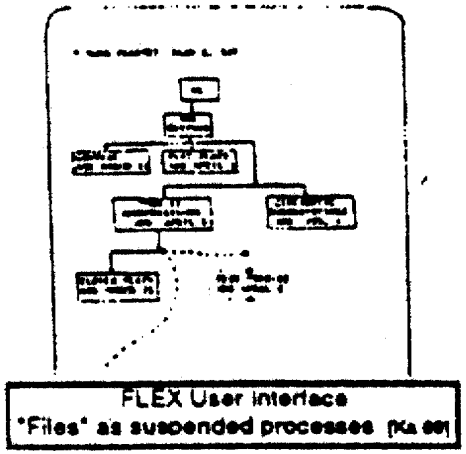
- Simula-like language
  ☛ objektorientierte Konzepte

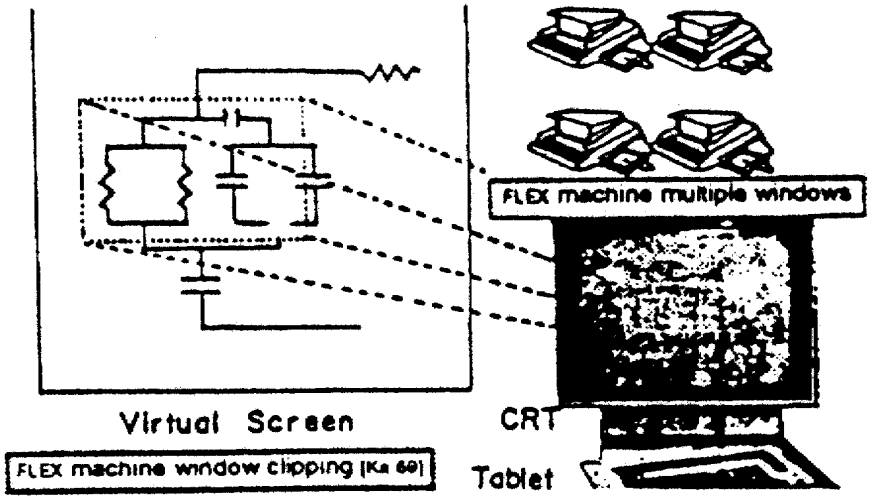
- einfacher Display-Editor für Text und Graphik

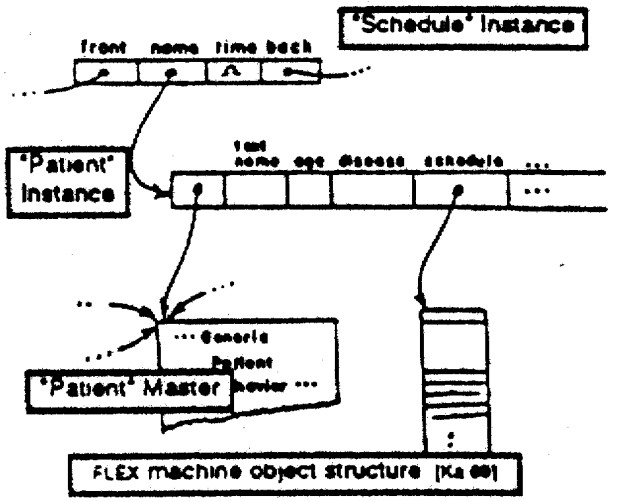
Alan Kay, “The Reactive Machine”, Ph. D. Thesis, Utah, 1969
'Dynabook'-Vision
Vision eines leistungsfähigen, portablen, persönlichen Rechners (Notebook)



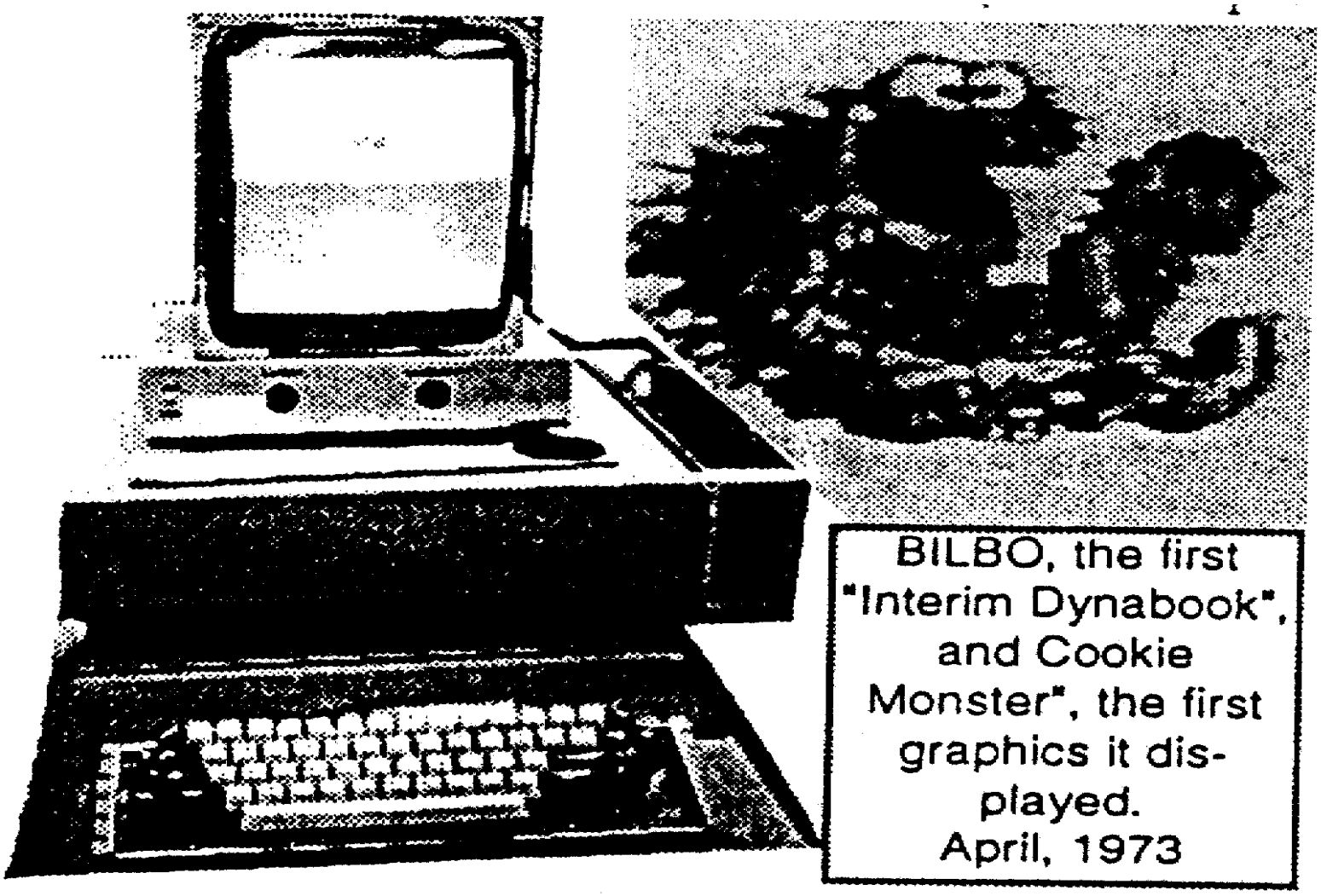
The FLEX Machine Self Portrait, ca 1968 [Ka 69]







(aus Kay 93)


BILBO, the first "Interim Dynabook", and Cookie Monster", the first graphics it displayed. April, 1973

(aus Kay 93)

1.4. Smalltalk-71

1970 Gründung des Xerox-PARC (Palo Alto Research Center)

1971 Gründung der Learning Research Group (notebook computer idea)

- Betrachtung jedes einzelnen Objektes als sein eigener Syntax-gerichteter Interpreter für Botschaften oder Anfragen, eine Funktion auszuführen.
- keine Implementation

➡ Metapher des Botschaftensendens (Kommunikation, Gespräch zwischen Objekten)

➡ Geheimnisprinzip

1.5. Smalltalk-72

- the first real Smalltalk
- 1000 Zeilenprogramm, das 3+4 (sehr langsam, aber richtig) auswertete.
- erste GUI mit überlappenden Fenstern
- eine Reihe von Anwendungen (Turtle-Graphic, Multimedia-Document Editor)

- Klassen, Instanzen
- keine Vererbung, keine Klassenhierarchie

Parallel dazu:

Entwicklung von Personal Workstations on Xerox PARC
Alto (1972) - first personal workstation

1.6. Smalltalk-74

Verbesserung von Smalltalk-72

OOZE - object-oriented zoned Environment

- virtual memory system

1.7. Smalltalk-76

- Sprachdesign bei Dan Ingalls
  - ☛ Thesen von Ingalls

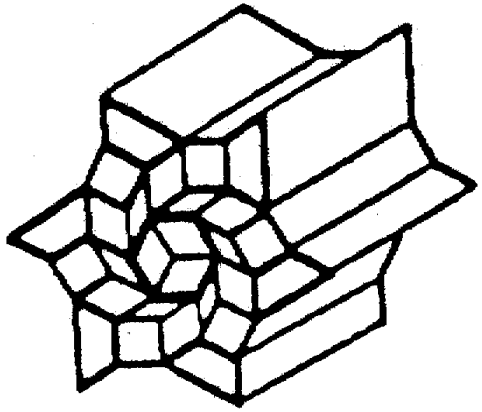
- Everything is an object.
  - ➡ Auch Klassen sind Objekte.
  - ➡ Metaclass *Class*
  - ➡ Jede Klasse ist Instanz dieser Metaklasse.

- Vererbung
  - ➡ Pseudovariablen self und super
  - ➡ Klassenhierarchie

- human readable syntax

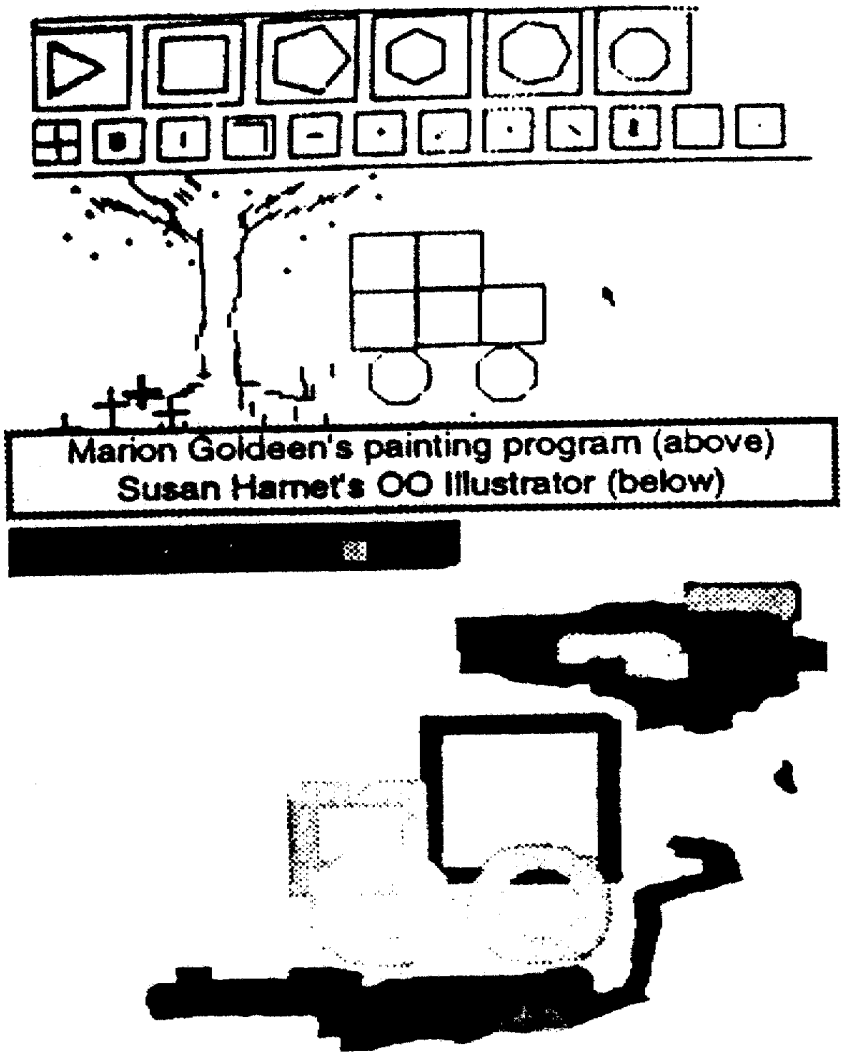
Smalltalk als Sprache für Kinder?

Kinder: ideale Testpersonen für Einfachheit und Verständlichkeit der Sprache



Tangram designs are created by selecting shapes from a "menu" displayed at the top of the screen. This system was implemented in Smalltalk by a fourteen-year old girl [Kay 77]

(aus Kay 93)



Marion Goldeen's painting program (above)
Susan Hamet's OO Illustrator (below)

(aus Kay 93)

Smalltalk-76

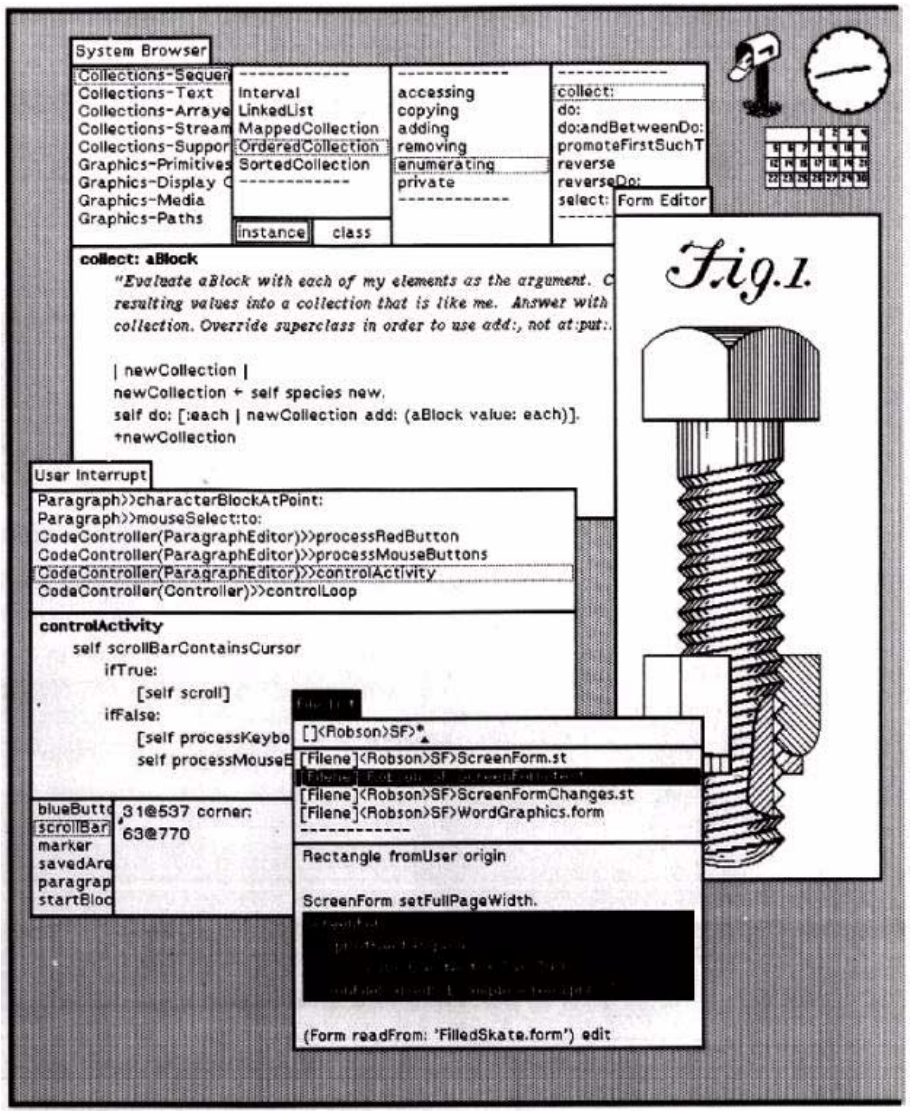
- Entwicklung der Benutzerschnittstelle des Smalltalk-Systems

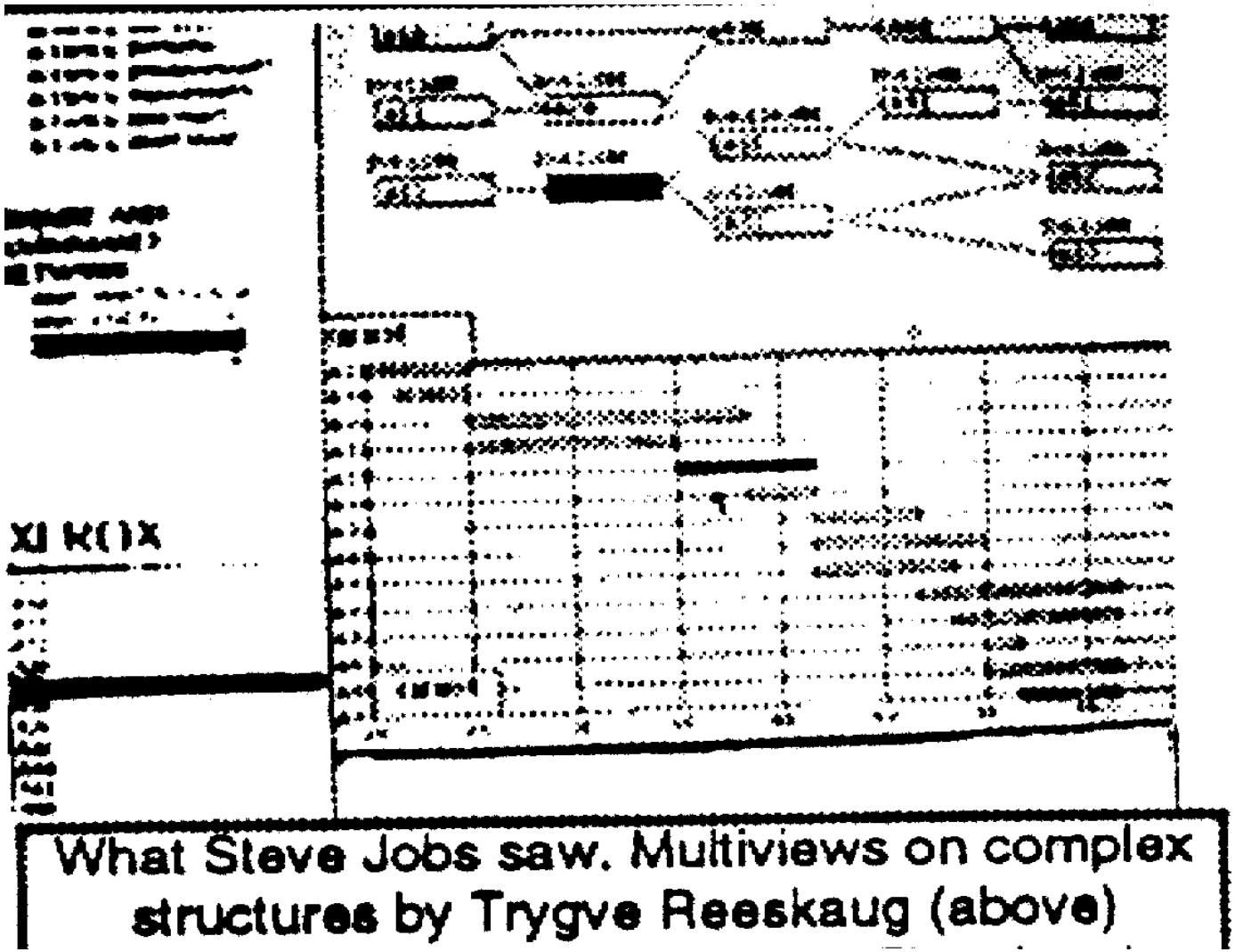
Ideen für die GUI

- Darstellen der Struktur der Software
- Techniken, um Fragen über die Struktur zu stellen
- Inkrementelle Möglichkeit für edit-compile-test Zyklen auf Methodenebene

Browser, Inspector, and Debugger (Larry Tesler)

☞ Ermöglichen einer interaktiven Arbeitsweise mit dem Computer


System Browser
Collections-Sequen
Collections-Text
Collections-Arraye
Collections-Stream
Collections-Suppor
Graphics-Primitives
Graphics-Display C
Graphics-Media
Graphics-Paths
Interval
LinkedList
MappedCollection
OrderedCollection
SortedCollection
instance
class
accessing
copying
adding
removing
enumerating
private
collect:
do:
do:andBetweenDo:
promoteFirstSuchT
reverse
reverseDo:
select:
Form Editor
Fig.1.
collect: aBlock
"Evaluate aBlock with each of my elements as the argument. C
resulting values into a collection that is like me. Answer with
collection. Override superclass in order to use add:, not at:put:.
| newCollection |
newCollection ← self species new.
self do: [:each | newCollection add: (aBlock value: each)].
↑newCollection
User Interrupt
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
controlActivity
self scrollBarContainsCursor
ifTrue:
[self scroll]
ifFalse:
[self processKeybo
self processMouseB
[]<Robson>SF>*
[Filene]<Robson>SF>ScreenForm.st
[Filene]<Robson>SF>ScreenFormChanges.st
[Filene]<Robson>SF>WordGraphics.form
Rectangle fromUser origin
ScreenForm setFullPageWidth.
(Form readFrom: 'FilledSkate.form') edit
31@537 corner:
63@770
scrollBar
marker
paragrap
startBloc



What Steve Jobs saw. Multiviews on complex structures by Trygve Reeskaug (above)

(aus Kay 93)

1.8. Smalltalk-80

- Xerox Workstations Dolphin und Dorado (1978)
- NoteTaker Projekt (Portabler Computer (3x8086)

- Bekanntmachung von Smalltalk außerhalb des PARC

- virtuelles Maschinenkonzept
  - ➡ leichtere Portierbarkeit
  - ➡ Smalltalk-System in Smalltalk geschrieben

- Metaklassenkonzept
  - ➡ klassenspezifische Erzeugungsmethoden

- Smalltalk-System Dictionary

- LOOM (large object-oriented memory)

- MVC, pluggable interfaces

Byte Magazine, August 1981 - Sonderausgabe über Smalltalk

1.9. Kommerzielle Smalltalks (1986-95)

ParcPlace Systems

ObjectWorks

Digitalk

Smalltalk/V

Knowledge System Corporation

Envy Smalltalk (später ObjectStudio)

IBM

VisualAge

1.10. Smalltalk-Hype (1995-1996)

- Probleme mit C++
- erfolgreiche Smalltalk-Projekte
- ausreichende Hardware-Performance
- IBM Smalltalk
- Zusammenschluß von ParcPlace und Digitalk
- weitere Smalltalk-Anbieter (Object-Arts, Smalltalk-MT)

1.11. Einbruch (1997-1999)

- Java-Hype
  - ☛ Quasi-Standard für OOPL
- Mißmanagement bei ParcPlace/Digitalk (später Objectshare)
- starke Verunsicherung

1.12. Renaissance (seit 2000)

- Probleme mit Java
- Prototyping-Fähigkeiten von Smalltalk
- Ähnlichkeit zwischen Java und Smalltalk (Bsp. Swing)
- Stabilisierung des Smalltalk-Markts

verschiedene kostenfreie Smalltalk-Systeme

- **Squeak** (Apple-Smalltalk)
- **Cincom Smalltalk** (VisualWorks und Objectstudio Noncommercial)
- **Smalltalk/X**
- **Visualage (30 Tage Evaluation-Version)**
- Dolphin
- Smalltalk Express

2. Die Grundphilosophie von Smalltalk

2.1. Entwicklungsziele bei Smalltalk

- Revolutionieren des Umgangs mit Computern
- Kontrolle und Verständnis aller wesentlichen Teile durch Benutzer
- Rechner als effektiv arbeitendes Werkzeug zur Problemlösung ohne komplizierte Benutzerführung

➡

1. Entwicklung einer (Programmier)sprache als Bindeglied zwischen den Modellvorstellungen des menschlichen Geistes und der Hardware.
2. Entwicklung einer Schnittstelle zwischen Benutzer und Rechner.
   Diese Schnittstelle sollte der zwischenmenschlichen Kommunikation entsprechen.

2.2. Was ist Smalltalk?

- eine *Programmiersprache*
- die Vision eines dynamisch anpassungsfähigen Entwicklungssystems
- eine komplette Programmierumgebung
- ein Werkzeug zum Entwerfen von Prototyp-Software-Varianten (interaktiv)

- die Realisierung der *Arbeitsplatzrechneridee*
- eine bestimmte Benutzerschnittstelle
- ein Betriebssystem
- ein Graphiksystem
- ein Datenbanksystem

2.3. Thesen von Ingalls („*Design Principles behind Smalltalk*“, Byte, August 1981)

- **Persönliche Verstehbarkeit**: Kontrolle und Verständnis aller wesentlichen Teile durch Benutzer.
- **Einheitlichkeit**: Eine Sprache sollte rund um ein *einheitliches Konzept* entwickelt werden, das bei allen möglichen Anwendungsfällen in gleicher Weise verwirklicht werden kann.
- **Objekte**: Eine Computersprache soll das Konzept von *Objekten* unterstützen.
- **Botschaften**: Berechnungen sollten als innere Fähigkeit von Objekten aufgefaßt und einheitlich über Botschaften aktiviert werden.

- **Modularität, Geheimnisprinzip:** Keine Komponente eines komplexen Systems sollte von den *inneren Details* irgend einer anderen Systemkomponente abhängen.
- **Polymorphismus**: Ein Programm sollte nur das Verhalten der Objekte spezifizieren, nicht aber deren Implementierung.
- **Klassifikation**: Eine Sprache muß Mittel zur Verfügung stellen, ähnliche Objekte zu *klassifizieren*. Es muß auch die Möglichkeit bestehen, neue Klassen von existierenden abzuleiten.
- **Speicherverwaltung**: Ein objektorientiertes Computersystem muß eine automatische Speicherverwaltung enthalten.
- **Prinzip der Reaktionsfähigkeit**: Jede Komponente, die für den Benutzer erreichbar ist, sollte sich in einer aussagekräftigen Form repräsentieren können.

2.4. Anwenden der Thesen auf Smalltalk

1. Everything is an *object*.

2. Objects communicate by sending and receiving *messages* (in term of objects).

3. Objects have their own memory (in term of objects).

4. Every object is an *instance* of a class (which must be an object).

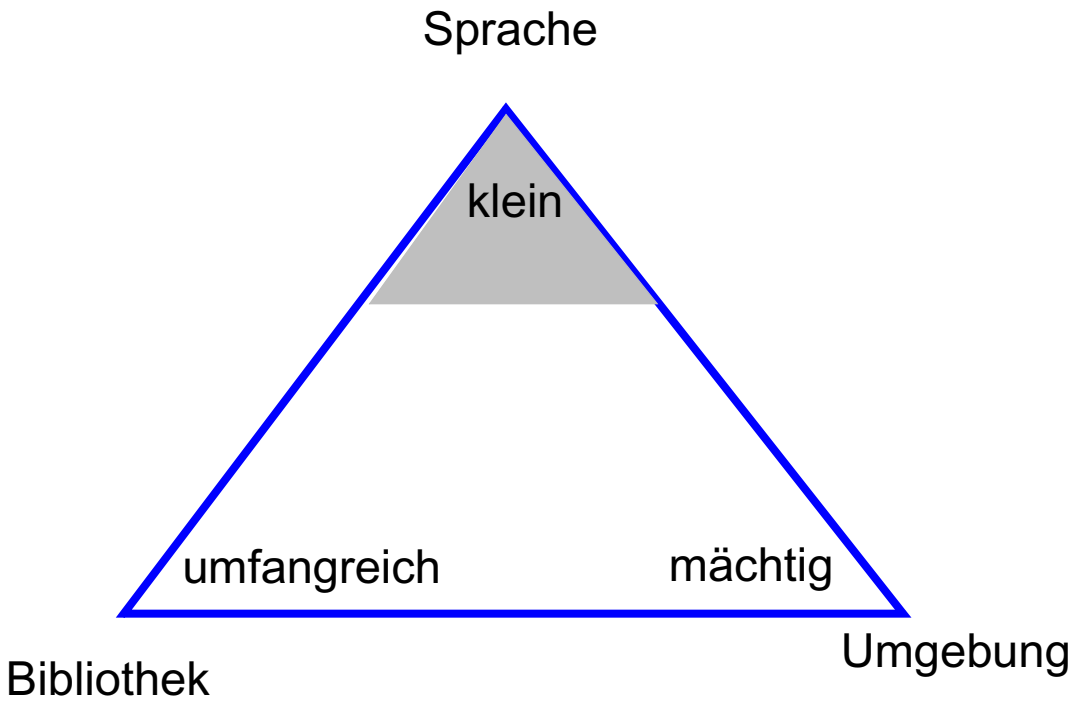
5. The class holds the *shared behavior* for its instances (in the form of objects in a program list).

6. To eval a program list, control is passed to the first object and the remainder is treated as its message.

7. Classes are organized into an *inheritance* hierarchy.

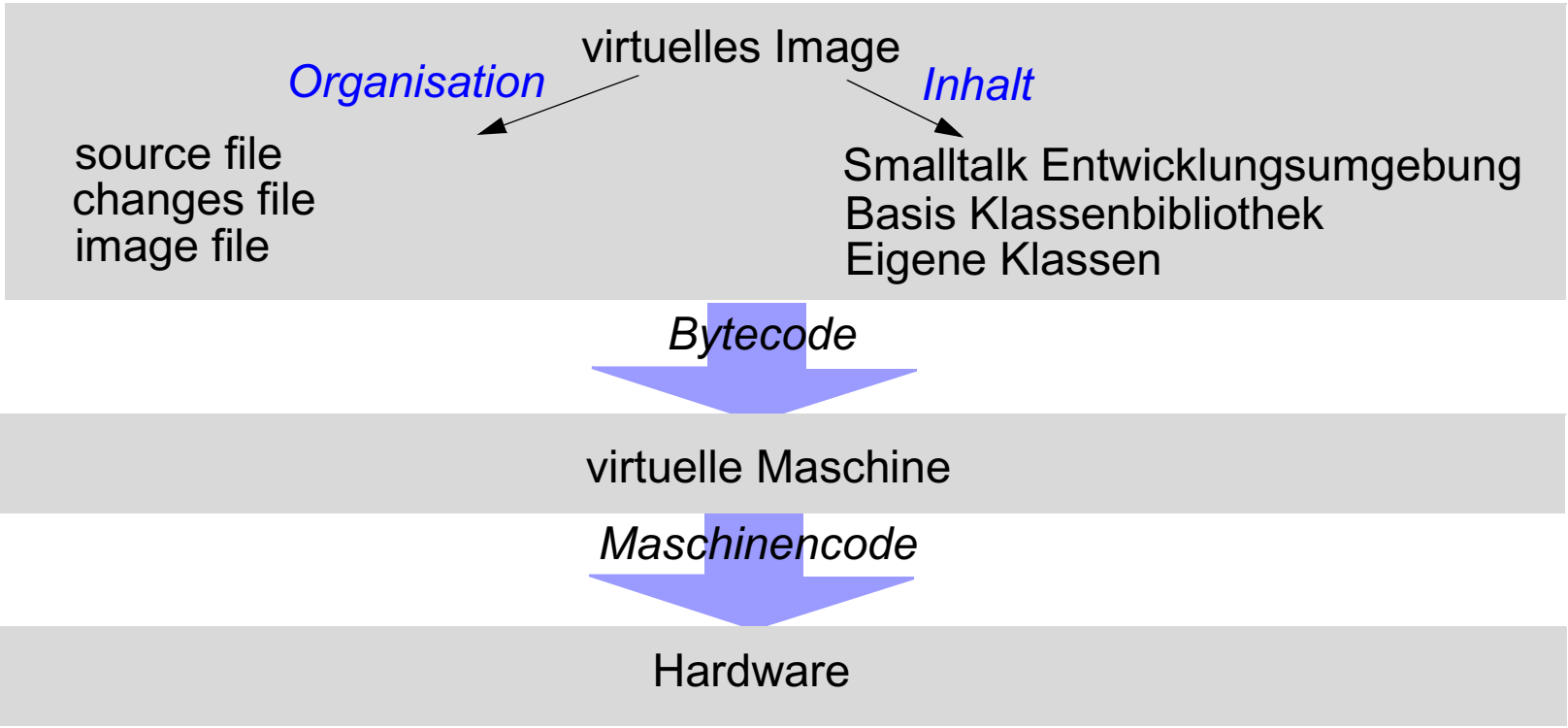
2.5. Sprachstruktur

Smalltalk = Programmiersprache + Klassenbibliothek + Entwicklungsumgebung



2.6. Virtuelles Maschinenkonzept

- nur *primitive* Methoden sind in Maschinensprache realisiert (kleiner Teil).
- *alle* anderen Methoden sind in Smalltalk selbst geschrieben.



➡ Um ein Smalltalk-System zu portieren, reicht es aus, auf dem Zielprozessor die virtuelle Maschine zu simulieren.

2.7. Problemlösen in Smalltalk

- Identifizieren der in der Aufgabenbeschreibung vorkommenden Objekte
- Einteilung in verschiedene Klassen
- Bereitstellen der Methoden, die die Kommunikation zwischen den Objekten ermöglichen

☞ Intergation und Erweiterung des bestehenden Smalltalk-Systems

☞ Wiederverwendung schon existierender Klassen (Objekte)

Inkrementelle Arbeitsweise

- schrittweise Erstellung eines Programms mit sofortiger Kontrolle und Testen
  - ➡ verwertbare Teillösungen
  - ➡ dynamischer Prozeß zwischen Entwerfen, Implementieren und Testen

Evolutionäres Programmieren

- Verwenden des bestehenden Systems und Erstellung eines Anwendungprogramms zur Lösung des Problems.

- Beobachten, wie sich das System bei der Problemlösung bewährt.
  Nachdenken über mögliche Verbesserungen.

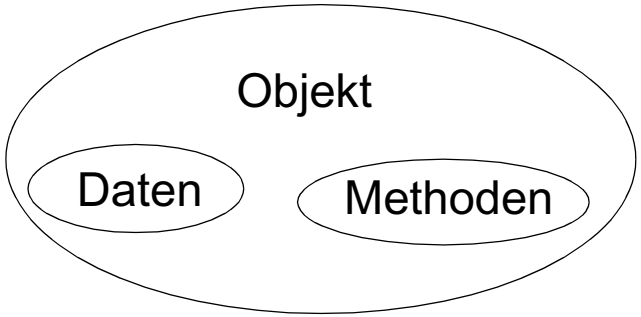
- Implementierung dieser Verbesserungen im System.
  - ➡ Neue Programmversion

- ☛ Sehr gute Prototyping-Fähigkeiten
- ☛ Experimentiermöglichkeiten
- ☛ Ausprobieren

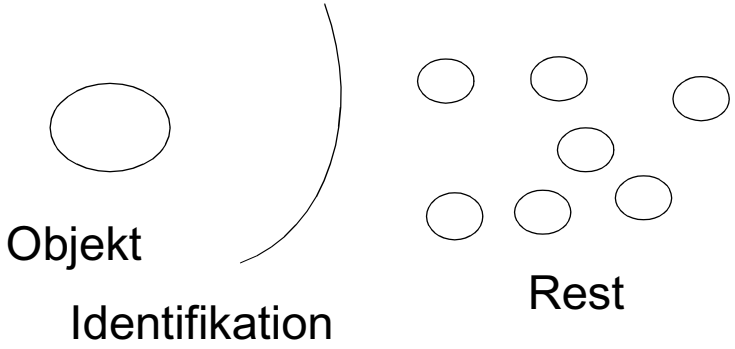
„Statt eines bit-zermalmenden Prozessors, der Datenstrukturen zerfetzt und vergewaltigt, haben wir (in einem objektorientierten System) eine Welt von Objekten mit guten Manieren, die sich höflich gegenseitig bitten, diverse Wünsche auszuführen. Der Austausch von Botschaften ist der einzige Prozeß, der außerhalb von Objekten durchgeführt wird. So soll es auch sein, da nur Botschaften zwischen den Objekten herumreisen dürfen.“ (aus Mittendorfer 1990)

3.1. Objekt

- Objekt = Variablen + Methoden
- Variablen sind nur Bezeichner (ungetypt)



- Identifikation eines Objekts (Anschauungsbereich)
  ➡ Trennung dieses Objekts vom Rest des Systems



(aus Mittendorfer 1990, S.22)

3.2. Objektorientiertes System

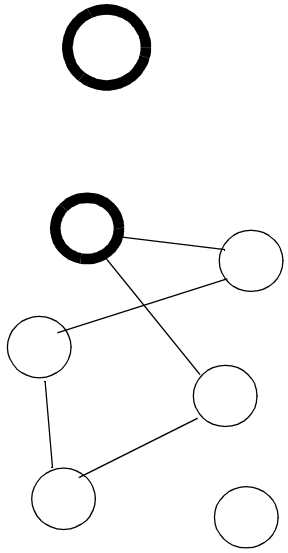
- Menge von Objekten

„Systemablauf“

1). Erzeugung eines Initialobjektes

2). Ausführen einer Startmethode dieses Objekts und Erzeugen neuer Objekte

3). Botschaften an diese neuen Objekte

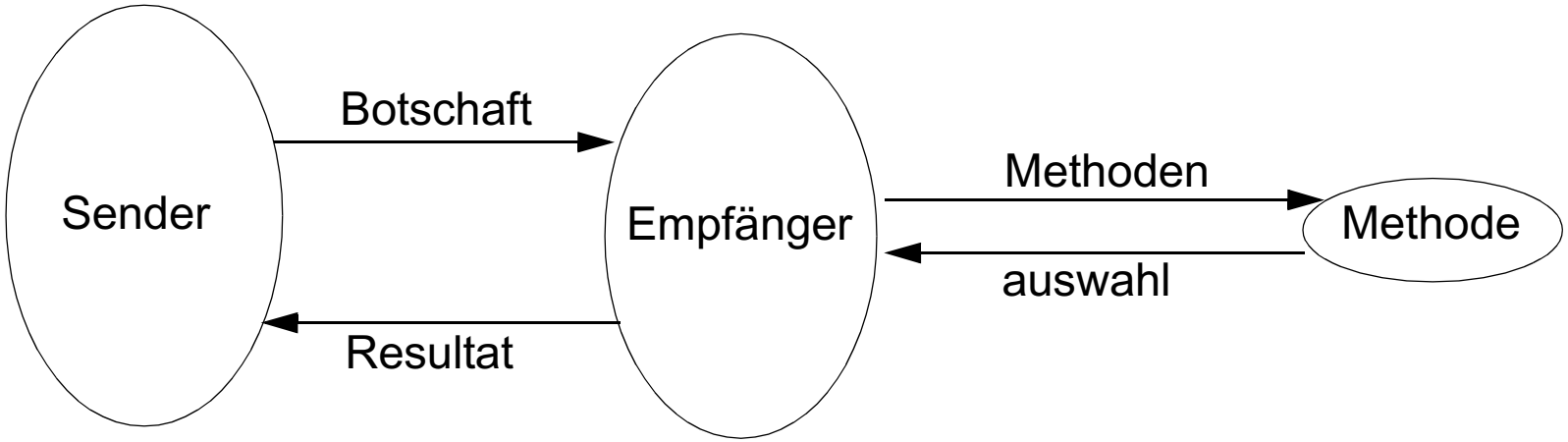


➡ schrittweises Wachsen und Schrumpfen des Objektbestandes

3.2. Botschaften

- Anforderung an ein Objekt, eine bestimmte Methode (Operation) auszuführen.

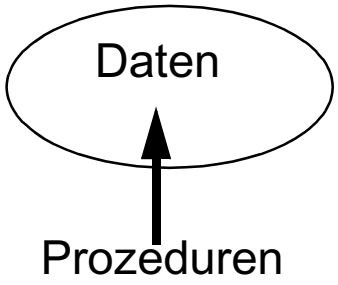
Botschaftenaustausch

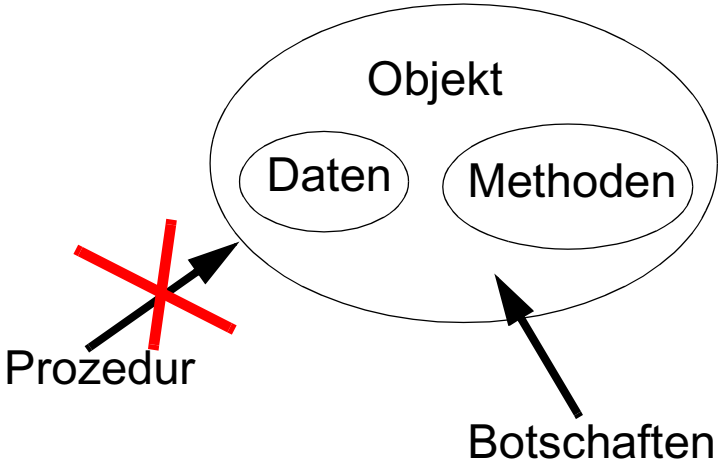


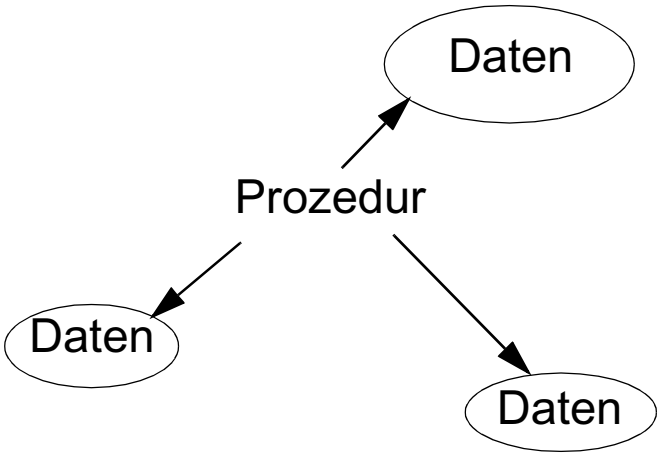
Fazit:

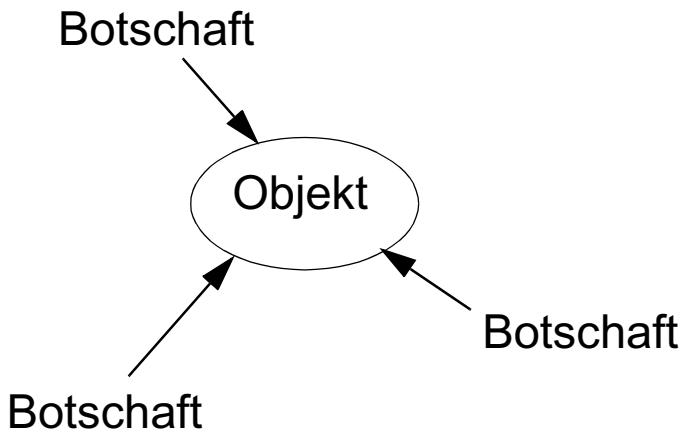
- Objekte reagieren nur auf Botschaften.
- Innere Details eines Objekts können nicht direkt geändert werden.
- Nur über eine Menge vordefinierter Botschaften (Botschaftenprotokoll) ist eine Manipulation möglich.

3.3. Prozedurale und objektorientierte Sprachen



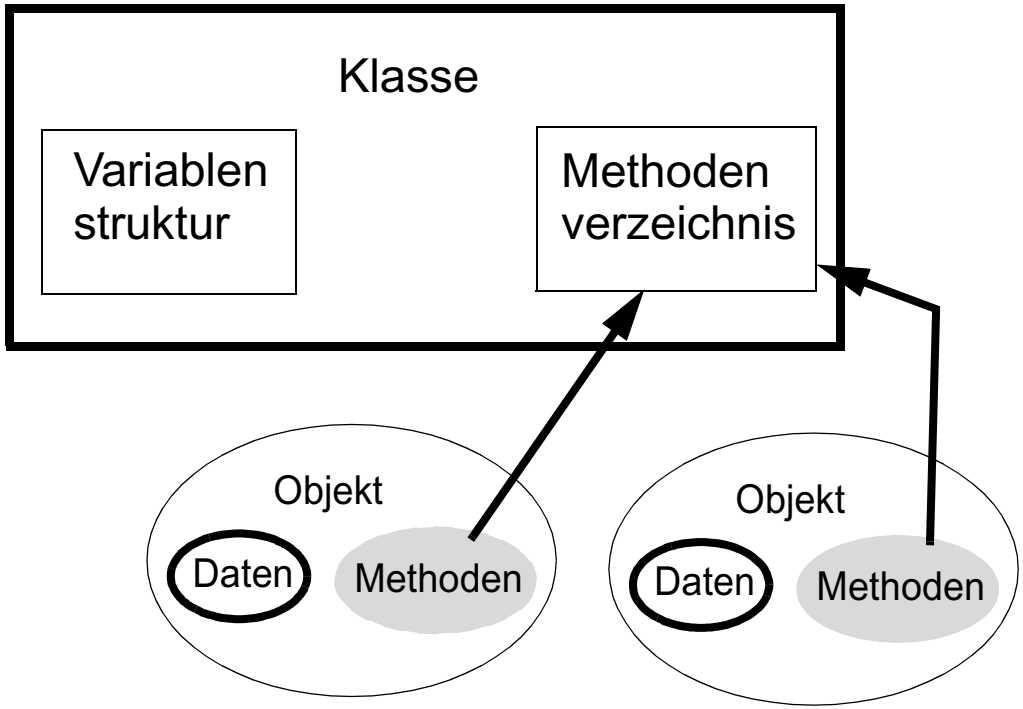






3.4. Klasse

- Zusammenfassung gleichartiger Objekte
  - ➡ Schablone für ihre Objekte



3.5. Vererbung in Smalltalk

Definition einer Klasse auf Basis einer existierenden Klasse

➡ Ableitung

➡ Erweiterung, Abänderung bestehender Klassen

➡ „Programming by Refinement“

- Alle Komponenten werden geerbt (Variablen und Methoden).
- Variablen dürfen nicht überschrieben (redefiniert) werden.
- Methoden dürfen überschrieben werden.
- Smalltalk unterstützt nur Einfachvererbung.

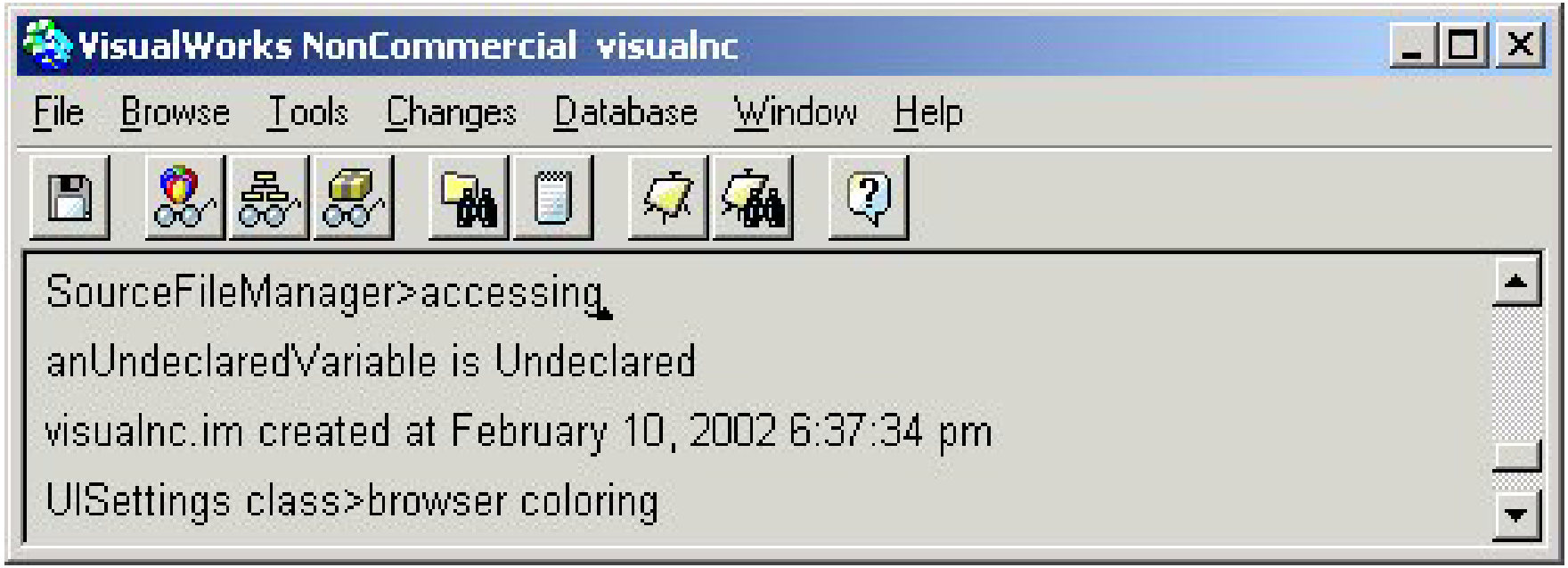
4. Einführung in VisualWorks

- www.cincom.com
- September 1999: Übernahme des Smalltalk-Geschäfts von Objectshare

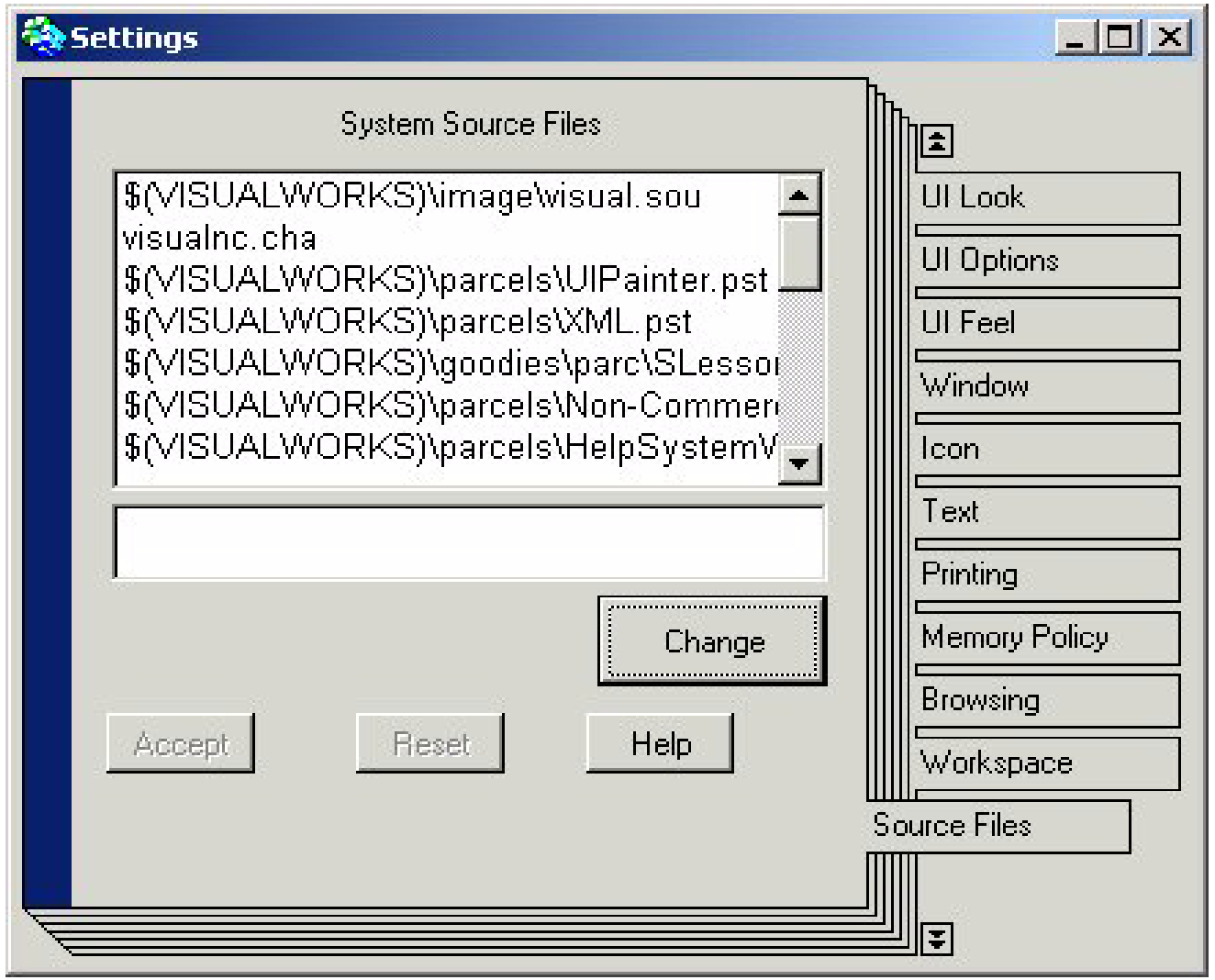
weiterführende Merkmale von Visualworks 5i.4

- Namespaces
- Web-Entwicklung - VisualWave
- Mehrbenutzer-Entwicklung - Store (vgl. zu Envy)
- Verteilte Systeme - Distributed Smalltalk
- viele Goodies: ODBMS GemStone; Refactoring Browser; S-Unit; DOME; Objectivity; Jun; HotDraw; ...

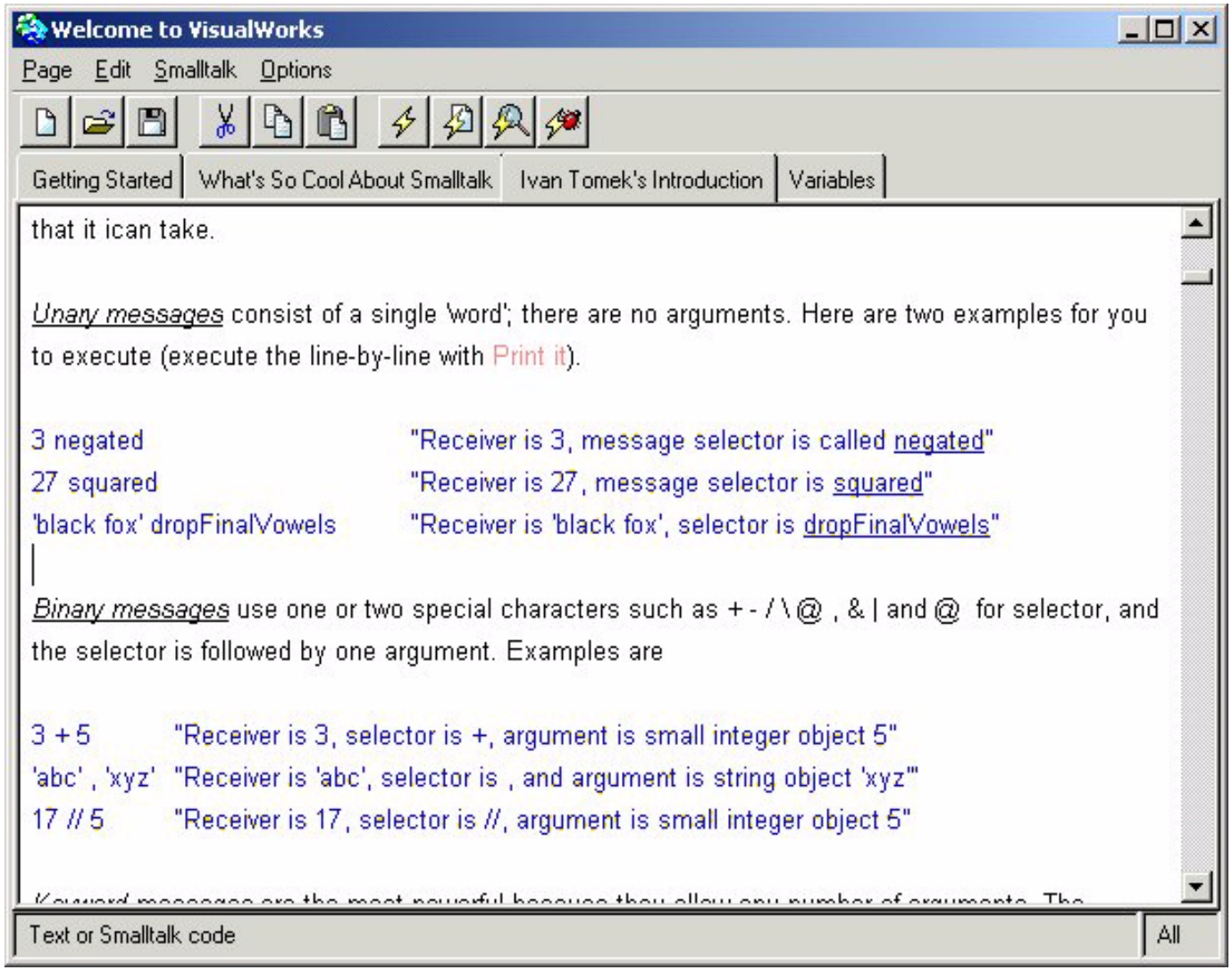
4.1. VisualWorks Launcher



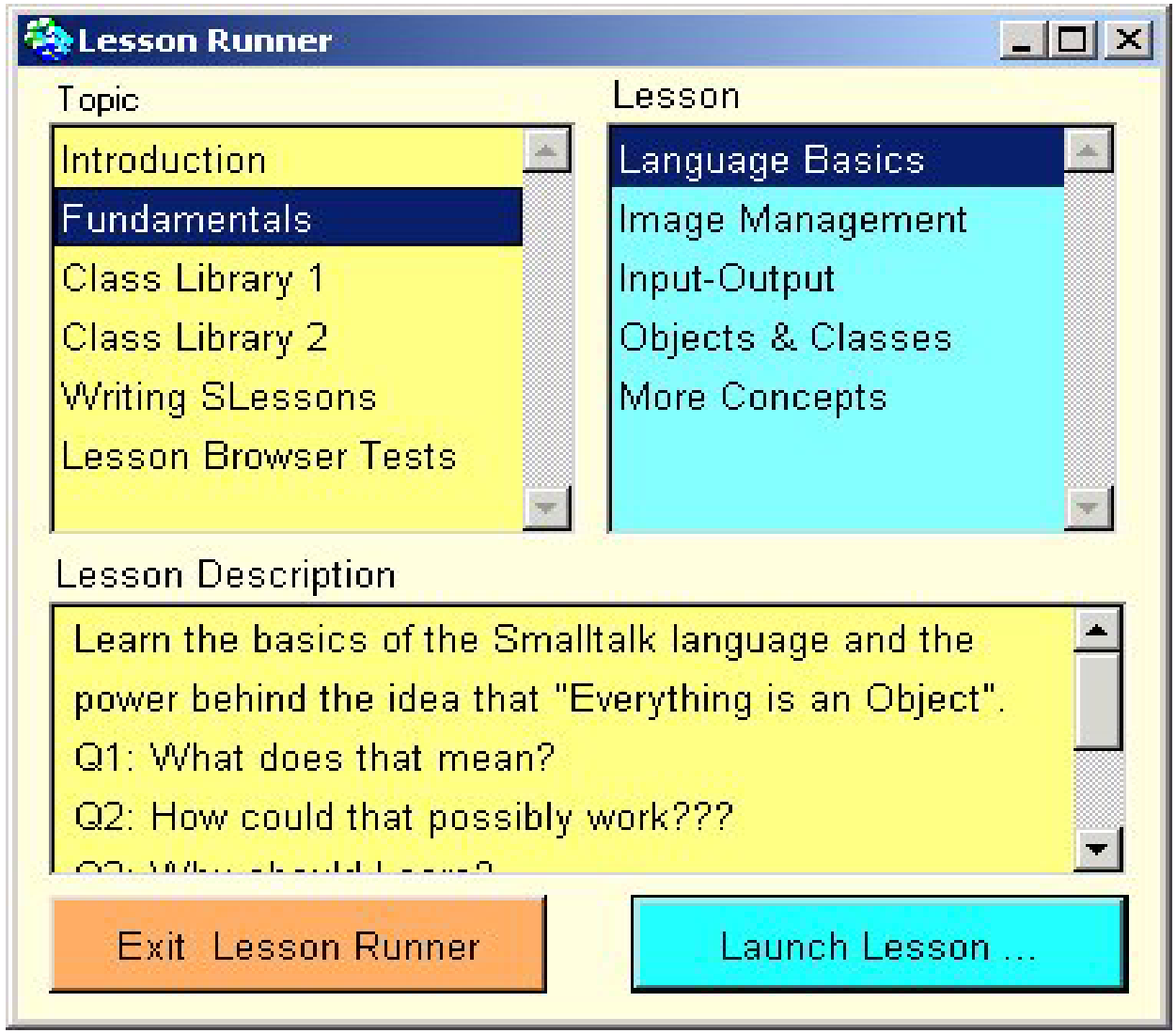
4.2. VisualWorks-Settings



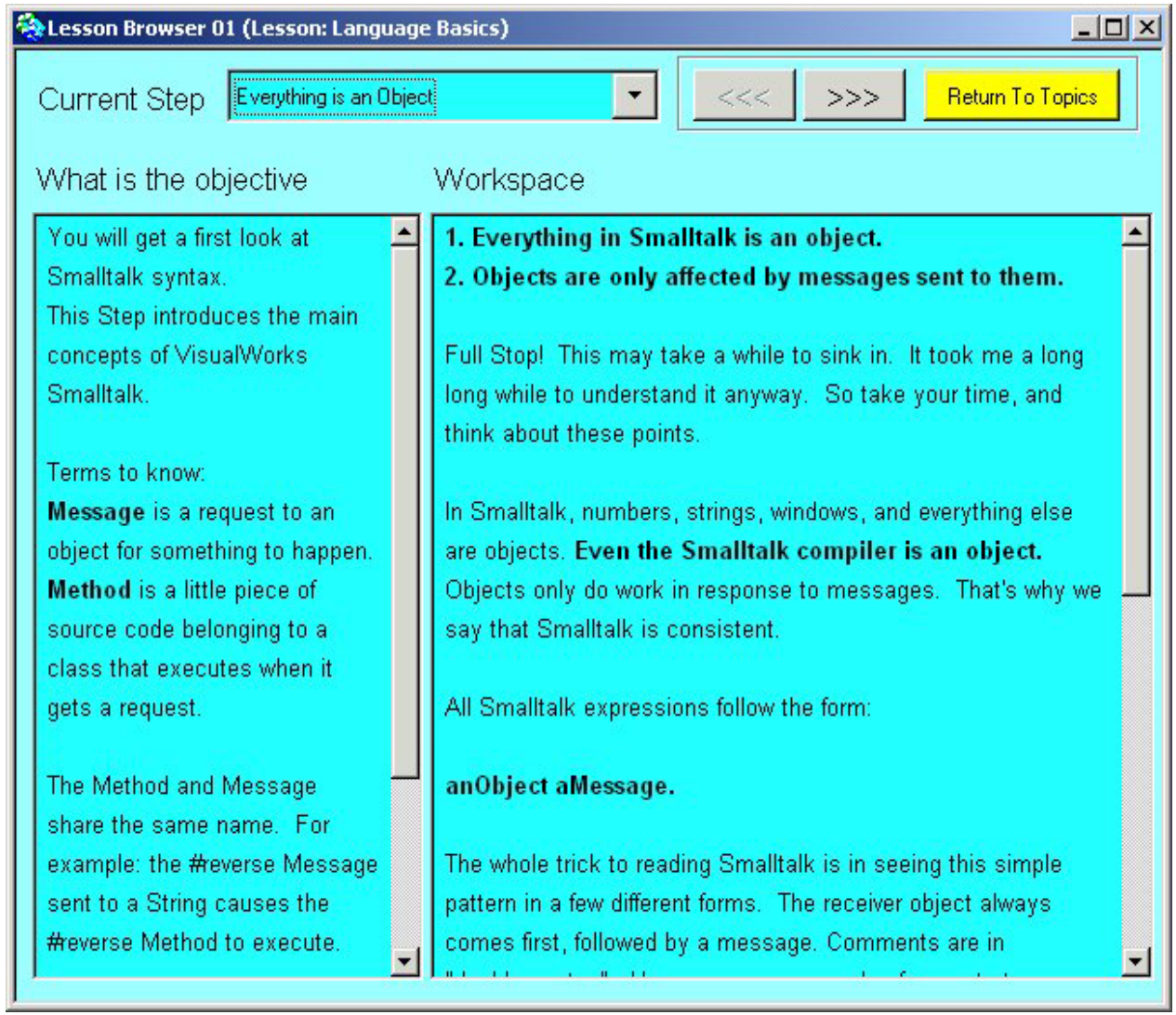
4.3. VisualWorks-Workspace



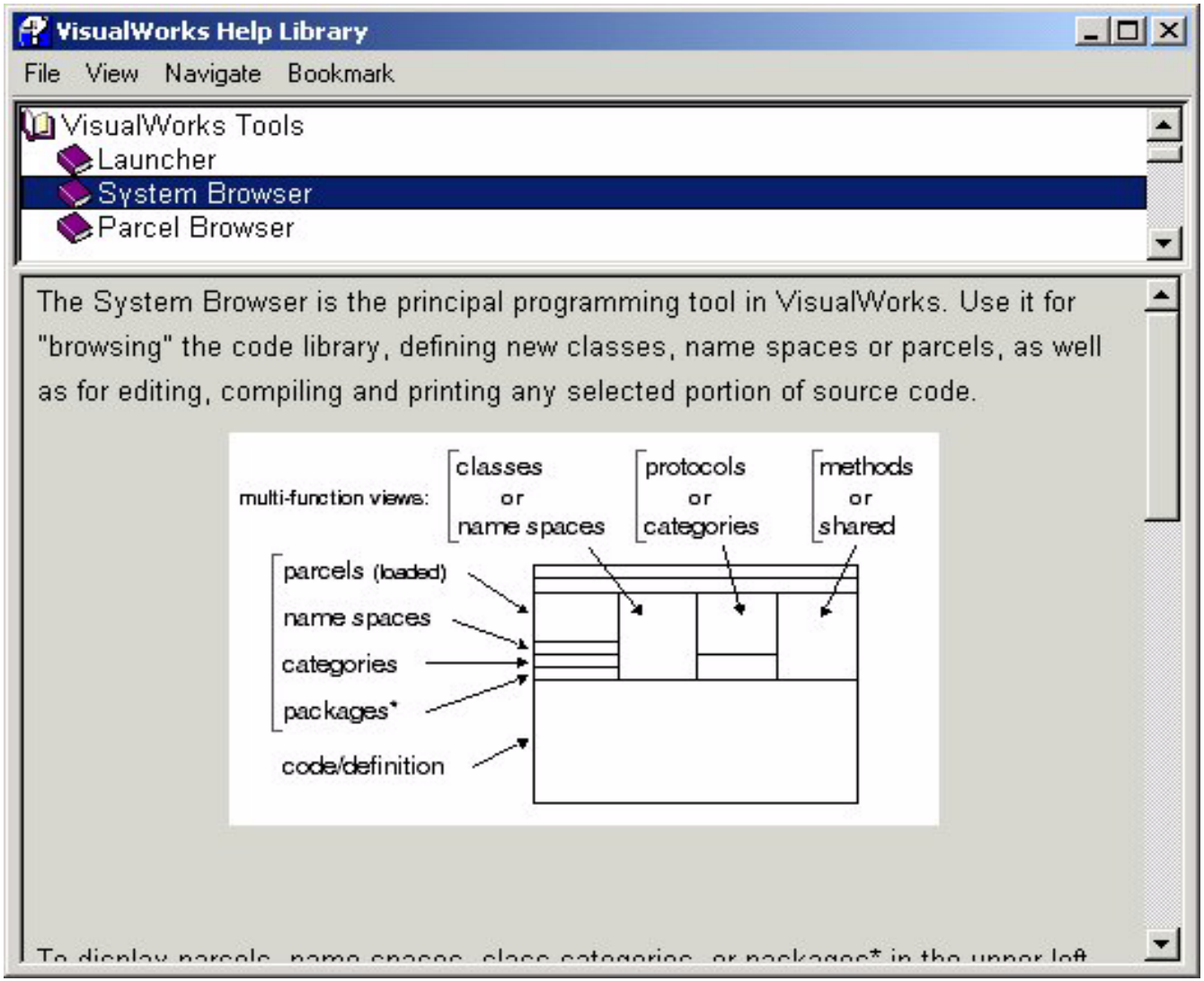
4.4. Hilfesystem: LessonBrowser (1)



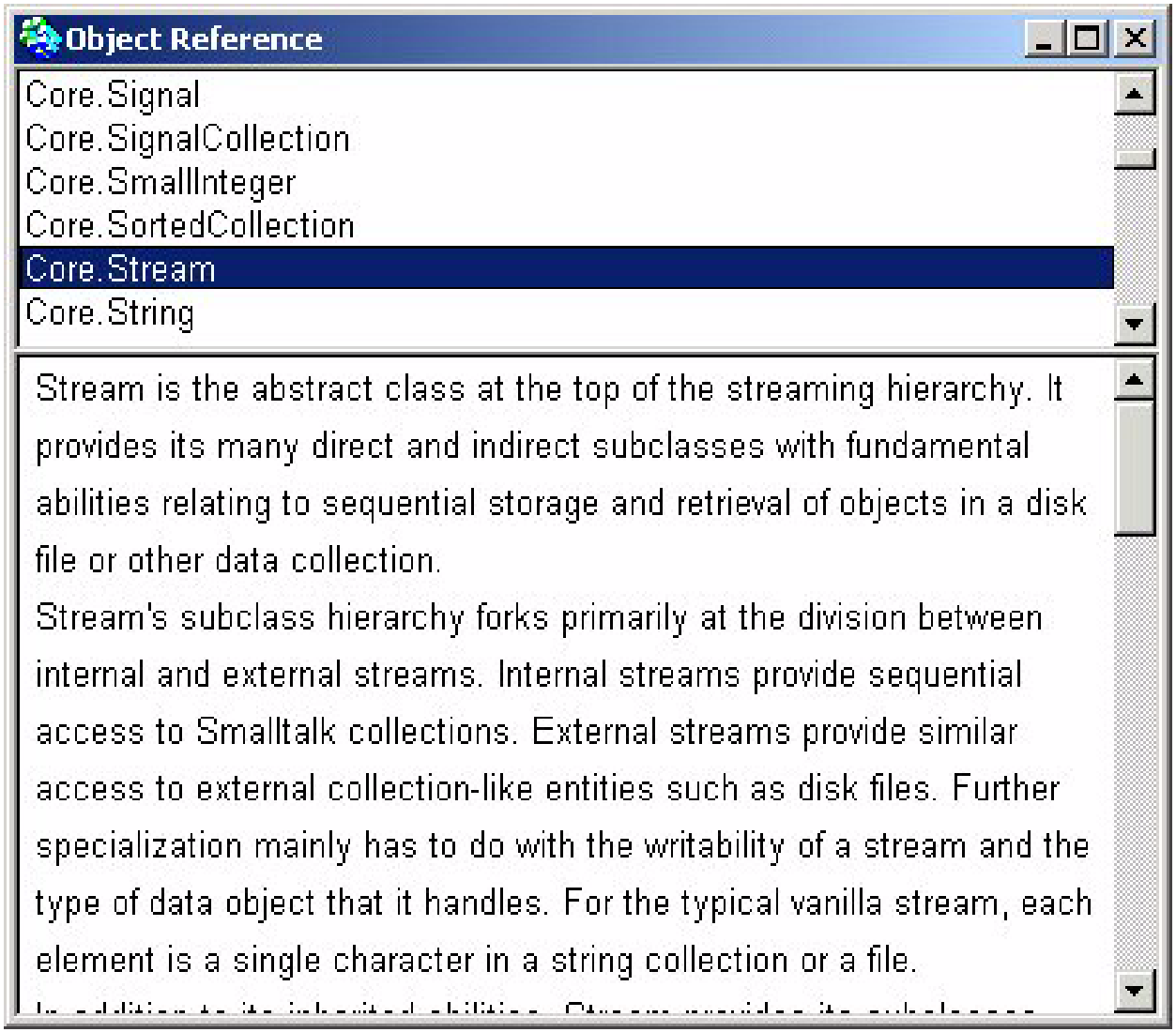
4.5. Hilfesystem: LessonTopics (2)



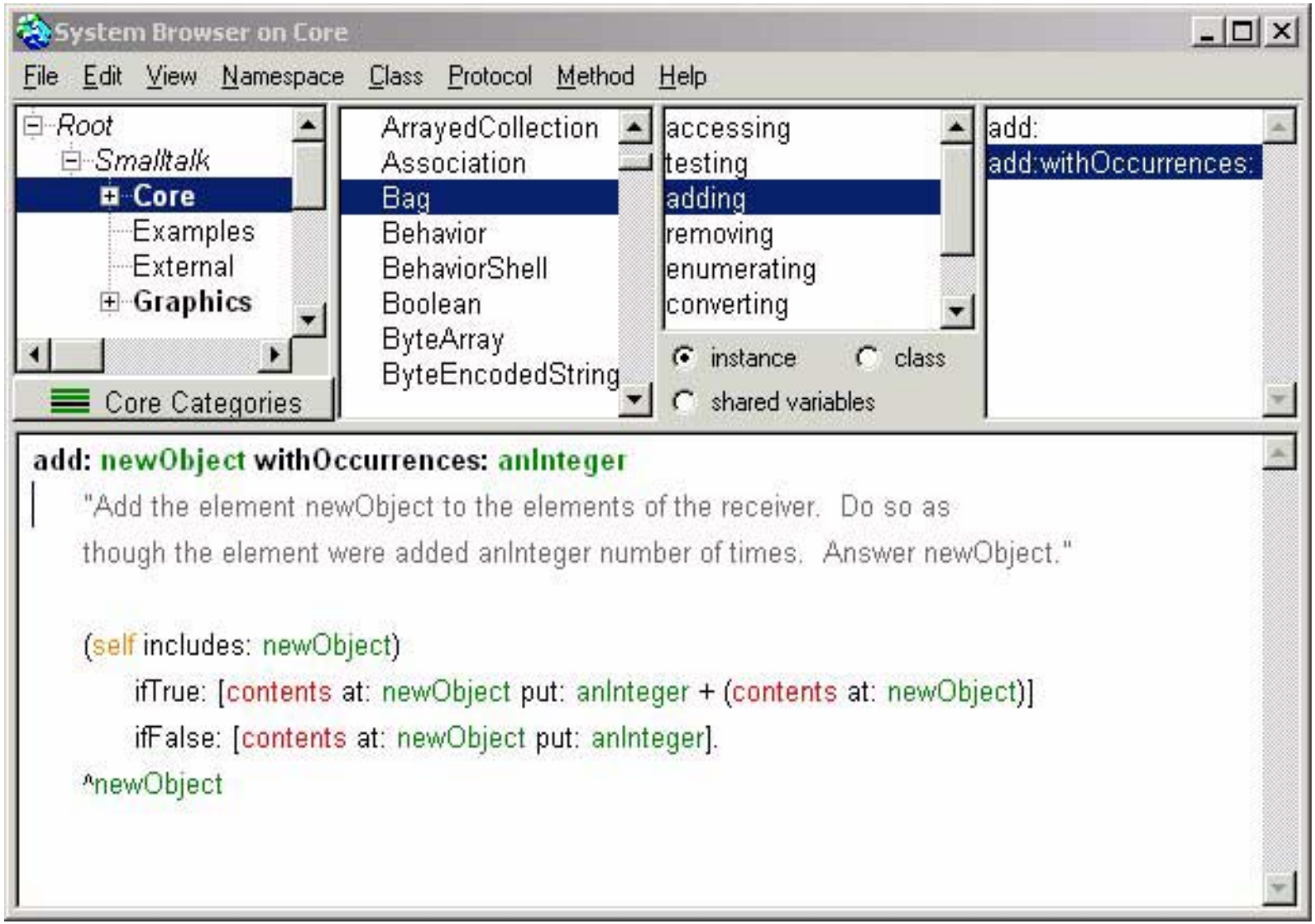
4.6. Hilfesystem-Cookbooks (3)



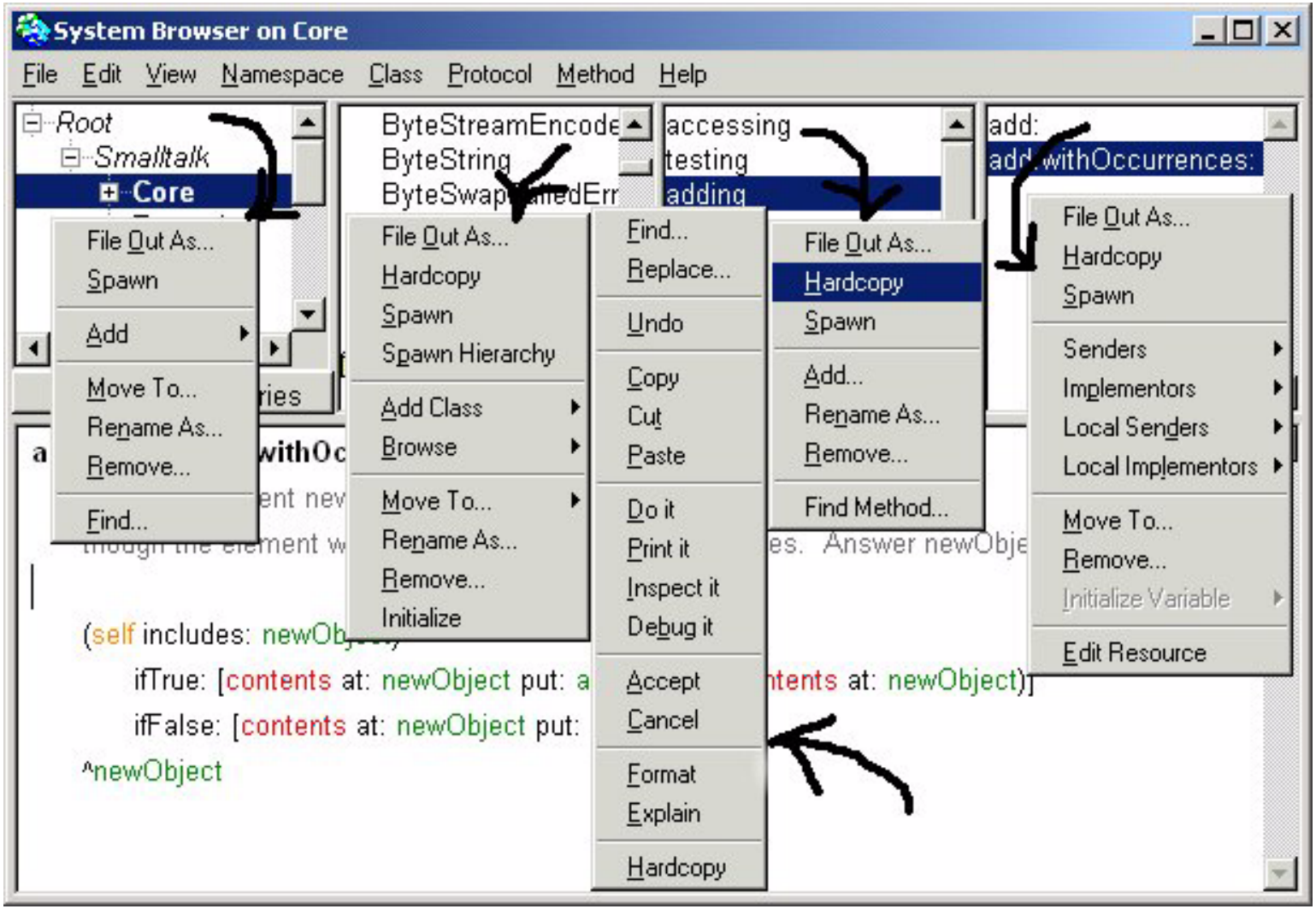
4.7. Hilfesystem-Object Reference (4)



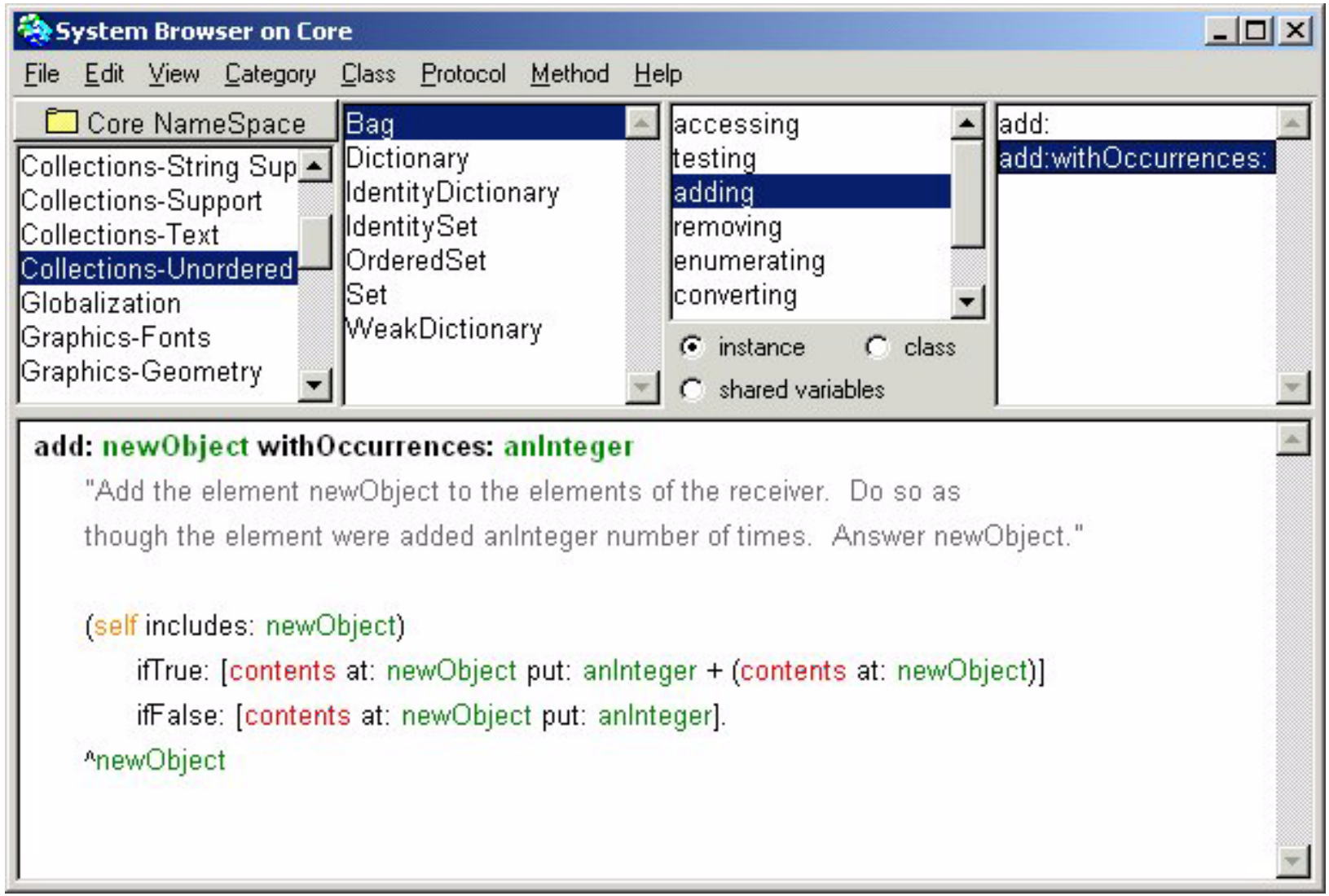
4.8. System Browser (1)



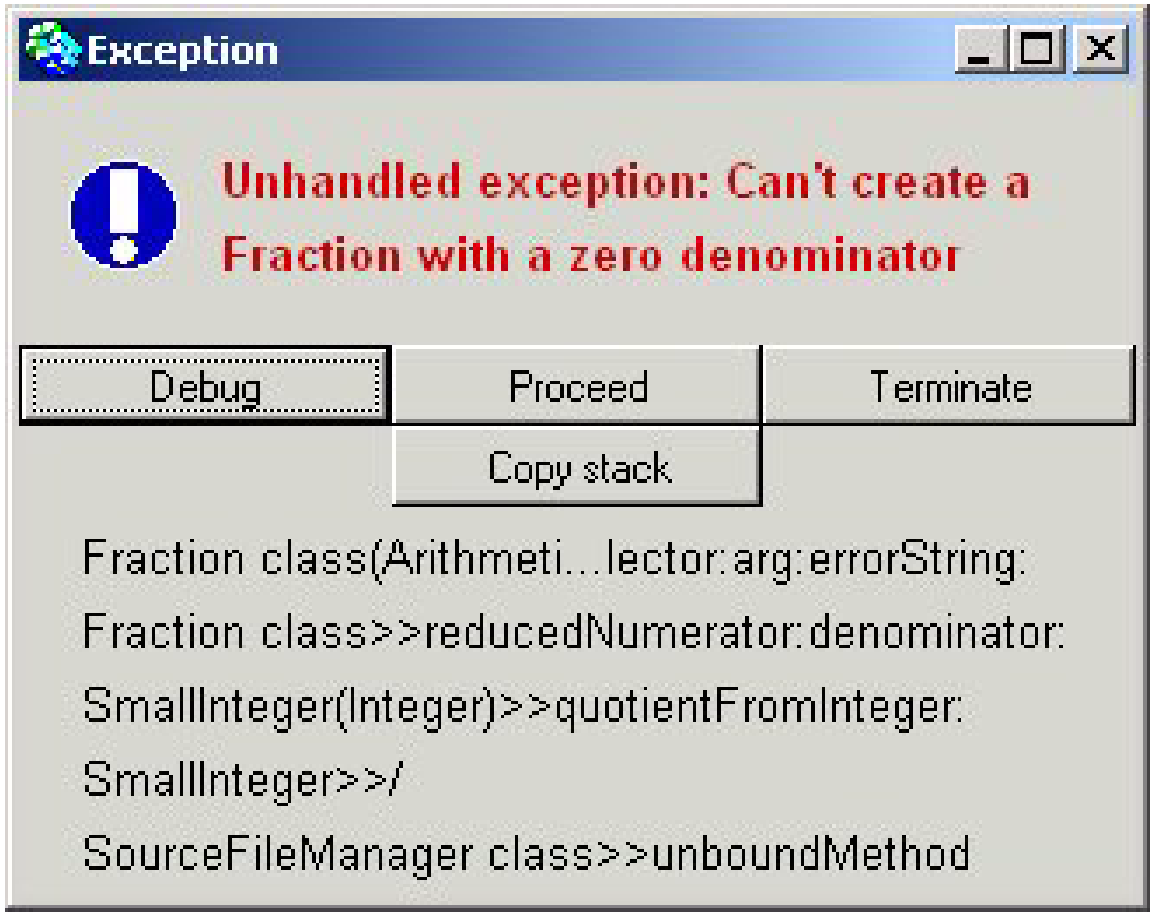
System Browser (2)



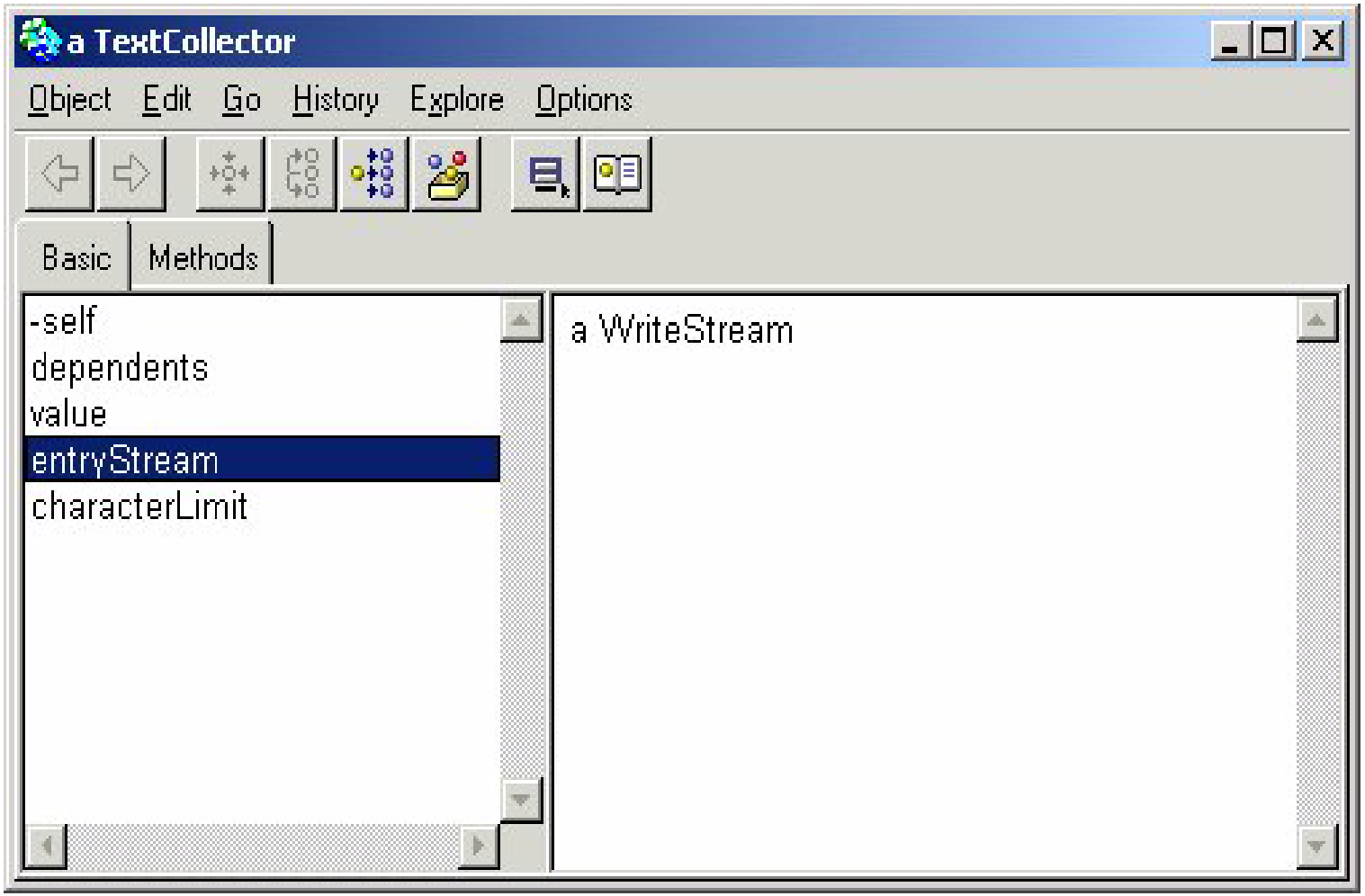
System Browser (3)



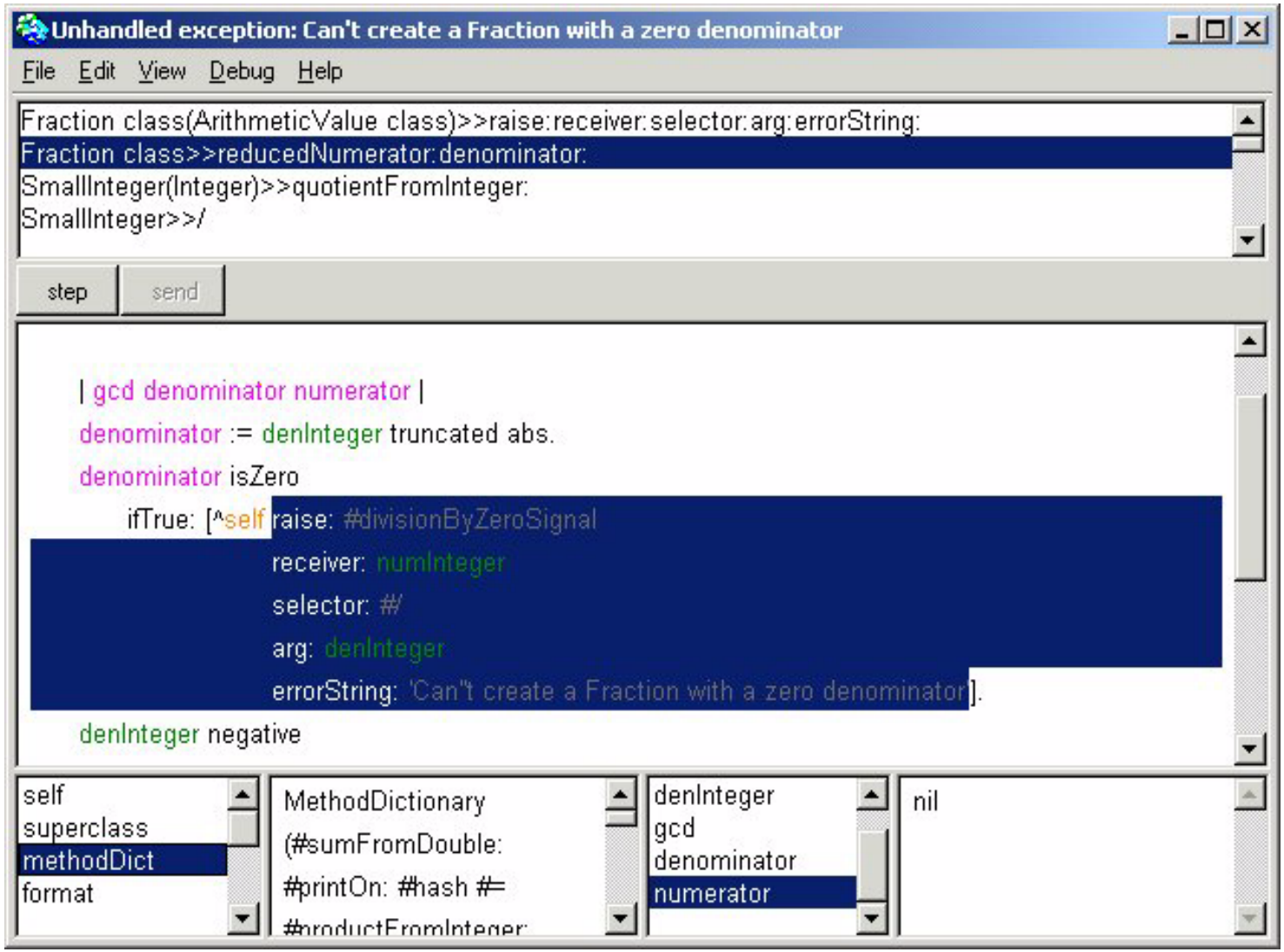
4.9. Exception



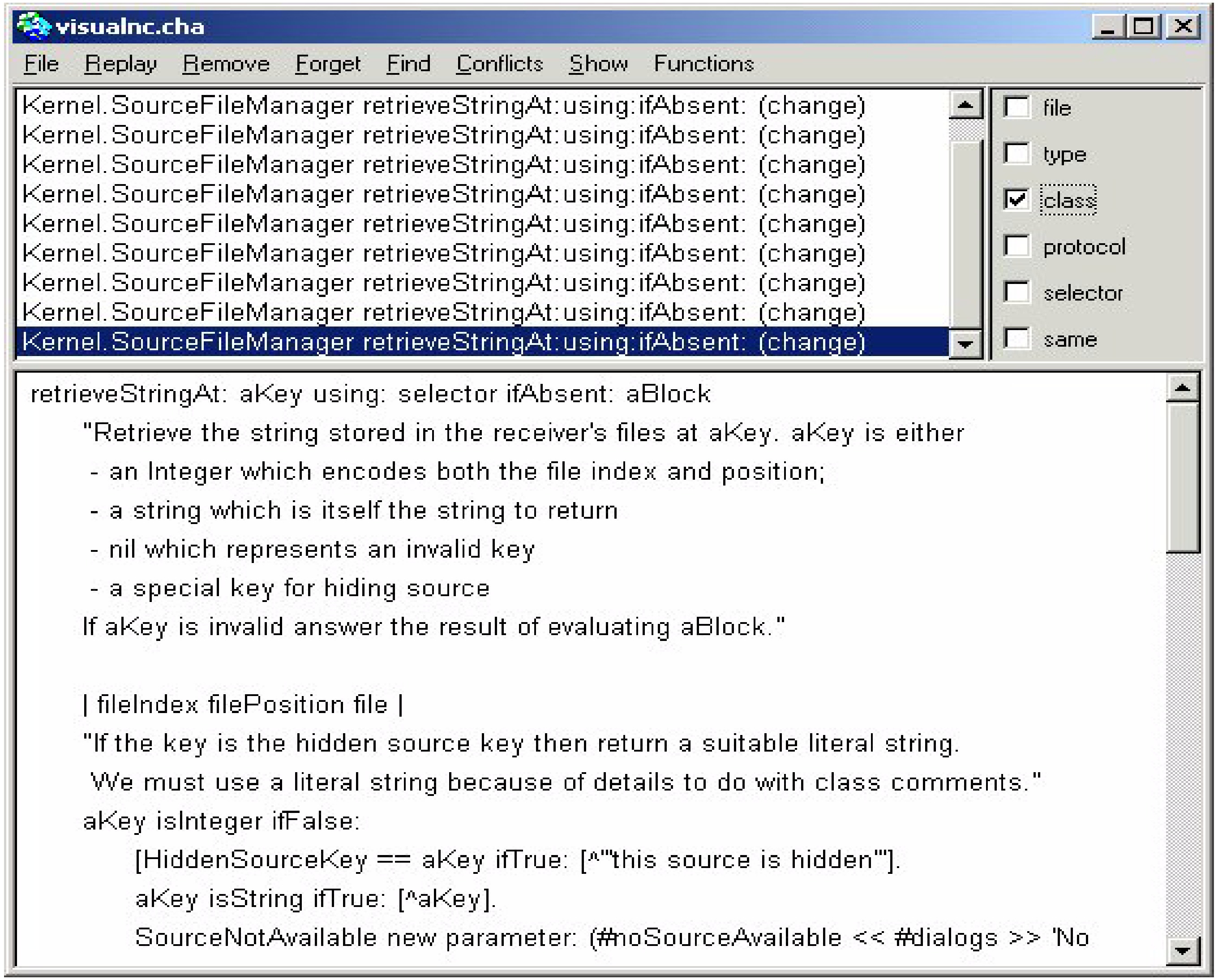
4.10. Inspector



4.11. Debugger



4.12. ChangeSet



5. Die Sprache Smalltalk

5.1. Botschaften

Botschaft = Empfänger Selektor Argument

- unäre Botschaften - kein Argument
  Bsp.: *Smalltalk allClasses*

- binäre Botschaften - spezielle Selektorsymbole (+, *, =, ==, ~= ...)
  Bsp.: *3 + 4, 4 == 4 copy*

- Keyword-Botschaften - mehrere Argumente
  Bsp.: *5 between: 3 and: 7*

Prioritätsregeln für zusammengesetzte Botschaften

- Klammern haben höchste Priorität.
  3+(4*5) -> 23; aber 3+4*5 -> 35

- unäre und binäre Botschaften werden strikt von links nach rechts abgearbeitet.
  3+4*5+6 entspricht ((3+4)*5)+6 -> 41
  Smalltalk allClasses size

- unäre Botschaften werden vor binären Botschaften abgearbeitet.
  3 + 4 sqrt -> 5

- binäre Botschaften werden vor Keyword-Botschaften abgearbeitet.
  4 raisedToInteger: 3+4 ->16384

- Keyword-Botschaften innerhalb zusammengesetzter Botschaften müssen geklammert werden.
  4 raisedToInteger: (128 log:2)

Botschaftenbinden - Das Suchen nach der richtigen Antwort !

Spätes Botschaftenbinden

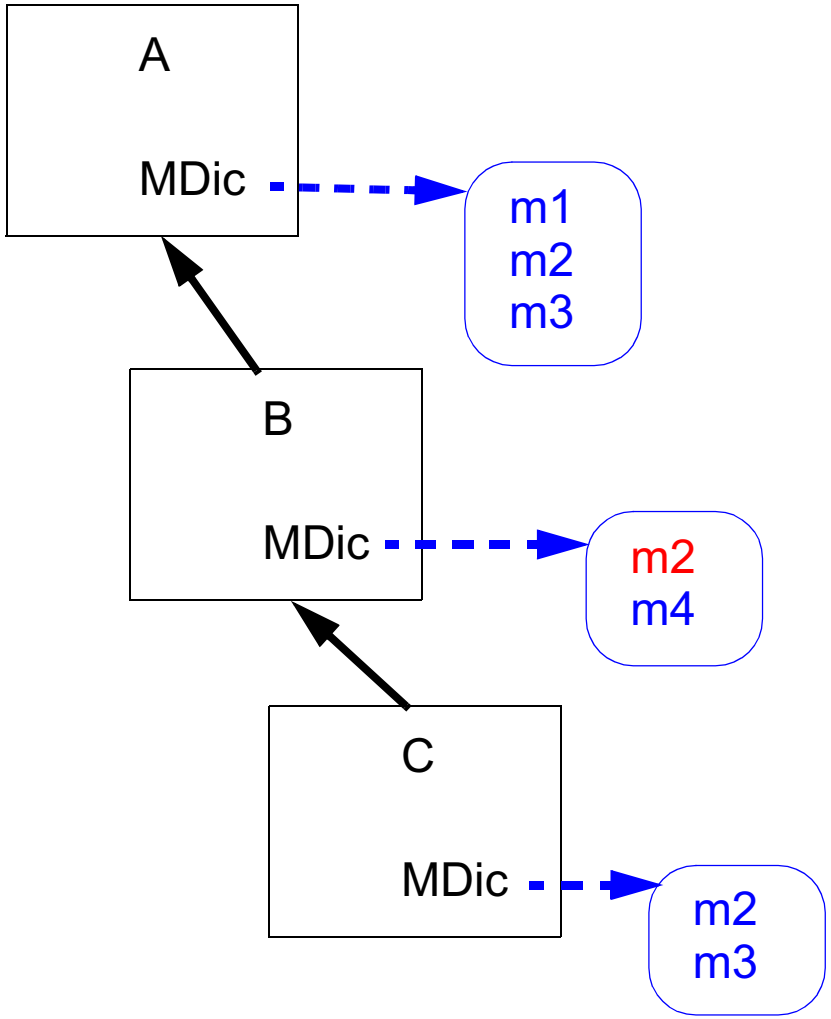
➡ die zu einer Botschaft gehörige Methode wird erst zur Laufzeit ermittelt.

Bsp.

aC m3

aC m1

aA m2



Rückgabe von Werten beim Botschaftensenden

Nach Ausführung der Botschaft muß dem Sender die Rückantwort zugeschickt werden.

- Gibt es keinen besonderen Rückwert, so wird der Empfänger an den Sender zurückgeschickt.

- Gibt es einen besonderen Rückwert, so wird dieser mit "^" -caret- gekennzeichnet.

Kaskadierung von Botschaften

statt:

```
anObject m1.
anObject m2.
anObject m3.
```

so:

```
anObject m1 ; m2 ; m3.
```

5.2. Pseudovariablen self und super

Pseudovariable self

- Möglichkeit, innerhalb einer Methode den Empfänger der Methode zu referenzieren.
- self steht für den Empfänger **selbst**.

Bsp.: Integer>>factorial

```
"#&Precondition: self >= 0"
"#&Postcondition:(self=0 and: [Result=1]) or:
                    [ Result= (Interval from:1 to:self)
                                inject: 1 into: [:subProduct :next | subProduct * next]]."

^ self < 2 ifTrue:[1] ifFalse:[self * (self -1) factorial]
```

Pseudovariable super

- Referenziert auch den Empfänger einer Botschaft (vgl. self).
- Die Suche nach der zugehörigen Methode beginnt aber nicht in der Klasse des Empfängers, sondern in der Oberklasse derjenigen Klasse, in der die Methode mit dem super-Aufruf implementiert ist.
- Ermöglicht die Ausführung der allgemeineren überschriebenen Methode

Beispiel für super

(aus Hoffmann, H.-J.: Smalltalk verstehen und anwenden, Hanser, 1987, S. 22 ff.)

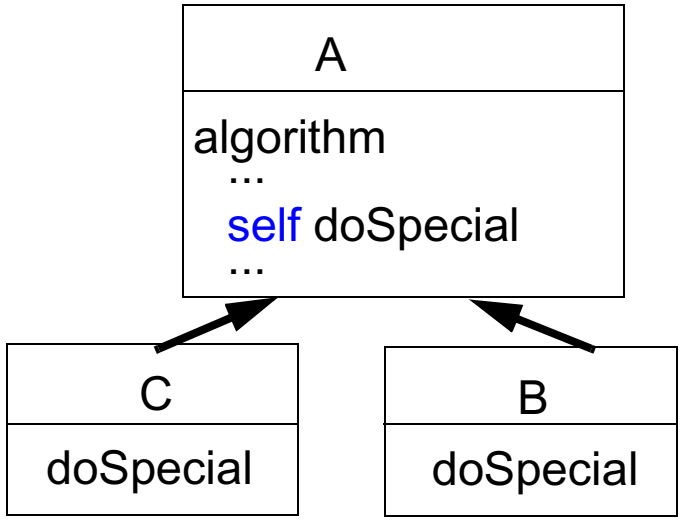
Object subclass: #**Eins** *methods* test ^ 1 result1 ^ self test	Eins subclass: #**Zwei** *methods* test ^ 2	Zwei subclass: #**Drei** *methods* result2 ^ self result1 result3 ^ **super** test	Drei subclass: #**Vier** *methods* test ^ 4
b1 := Eins new. b1 test. b1 result1.	b2 := Zwei new. b2 test. b2 result1.	b3 := Drei new. b3 test. b3 result2. b3 result3.	b4 := Vier new. b4 result1. b4 result2. b4 result3.

Verwendung von self und super mit dynamischen Binden

Definieren des allgemeinen Algorithmus
in der Oberklasse.
- gilt für gesamten Klassenteilbaum

Aufruf der speziellen Methoden über self.

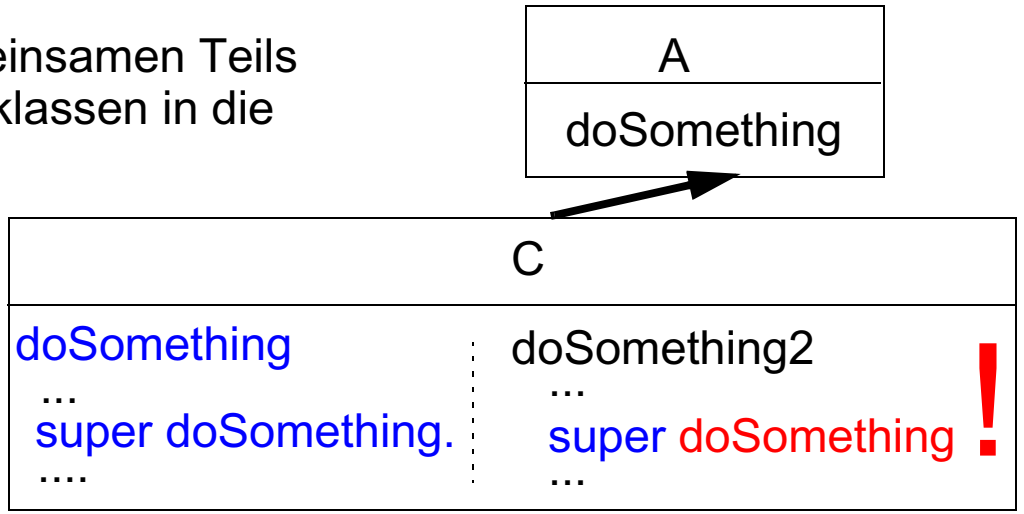
Parameterisieren der allgemeinen
Methode mit den speziellen Methoden.



Herausfaktorisieren des gemeinsamen Teils
einer Methode aus den Unterklassen in die
Oberklasse.

--> Einfrieren dieses Teils

Aufruf innerhalb der
Unterklassen mit super.



5.3. Referenzsemantik

| a b|

a := #(1 2 3).

b := a.

b inspect. -> #(1 2 3)

b == a -> true

b := a copy.

„Gleichheit“
b=a. -> true

„Identität“
b==a. -> false.

b := a.

b at:2 put:’Smalltalk’.

a inspect. -> #(1 ’Smalltalk’ 3)

„Aliasproblem“

5.4. Blöcke

- Zusammenfassung einer Reihe von Anweisungen
  ➡ Auch Blöcke sind Objekte.

- Ausführen des Blocks, sobald er die Botschaft value erhält.

Bsp.
```
|b|
b := [ Transcript show: 'test'].
Transcript show:' jetzt '.
b value.
```

„Blöcke mit Blockvariablen“
```
b:= [:i :j| |a| a:=5. a+i+j]. b value:10 value:20.
```

Blöcke sind:
- namenlose Funktionen (Lambda-Kalkül)
- Definitionskontext = Umgebung des Lambda-Ausdrucks (Binden freier Variablen)
- ^ in Blöcken führt zum Rücksprung in den Definitionskontext.

5.5. Kontrollstrukturen

- Nur := (Zuweisung) ist keine Methode.
- Alle anderen Kontrollstrukturen sind aus Methoden und Blöcken konstruiert.

Bedingungen

max := a<b ifTrue:[b] ifFalse:[a].

Bedingte Schleifen

[a<b] whileTrue:[a:=a+a].
[self resetWindows. (Delay forSeconds:2) wait. true] whileTrue
zusätzlich noch whileFalse:

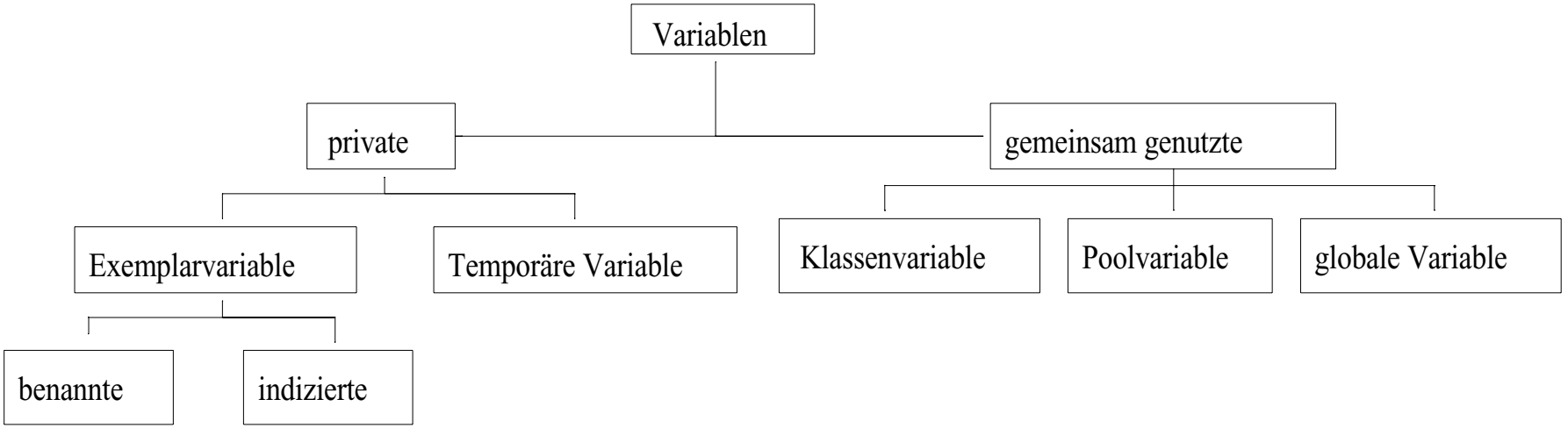
Zählschleifen

100 timesRepeat:[„do something“]
1 to: 100 by:2 do:[:i| s := s+i].

(1 to: 100) do:[:i| s:= s+i].
weitere Collection-Iteratoren: collect:, select:, reject:, inject:into:

5.6. Variablen

- In Smalltalk sind Variablen Behälter für Objekte.
- Objekte selbst sind Zeiger auf Speicherplätze.
- Variablen sind untypisiert.



- Klassenvariablen, *Pool Dictionaries* und globale Variablen werden in VisualWorks Release 5i auf *shared variables* von *Namespaces* abgebildet.

5.7. Metaklassen

Wissen:

- Alle Komponenten des Smalltalk-Systems sind Objekte.
- Botschaften dürfen nur an Objekte geschickt werden.

Problem:

- Zur Erzeugung und Initialisierung von Objekten wird eine Botschaft an Klassen geschickt !!!

➡ Je nach Art des Empfängers unterscheidet man Exemplarmethoden und Klassenmethoden.

- Exemplarmethoden beschreiben die Fähigkeiten der Objekte einer Klasse, auf Botschaften zu reagieren.

- Klassenmethoden dienen z.B. zum Erzeugen eines Exemplars der Klasse (Methode *new*).

➡ Um die Einheitlichkeit beizubehalten, müssen auch die Klassen selbst Exemplare bestimmter Klassen sein.

➡ Diese Klassen bezeichnet man als Metaklassen.

➡ Ihre Methoden sind die Klassenmethoden aus dem Class-Browser.

Seit Smalltalk 80 existiert für jede Klasse eine zugehörige Metaklasse.

➡ eigene Klassenmethoden für jede Klasse
- spezielle Erzeugungsmethoden für ihre Exemplare.
- Ressourcen

➡ Initialisierung von Klassenvariablen durch Klassenmethoden (initialize)

➡ Vererbungshierarchie der Metaklassen ist parallel zur Vererbungshierarchie der Klassen.
- Auch Metaklassen erben Methoden ihrer Oberklassen!

5.8. Klassendefinition

- Klassen werden durch Senden einer Botschaft an ihre zukünftige Oberklasse erzeugt (Smalltalk-80 und VisualWorks bis Release 3.0).

Methodenprotokoll *Class>>subclass creation*

```
subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat
    "This is the standard initialization message for creating a new class as a subclass
    of an existing class (the receiver)."
```

Klassenformen: *subclass, variableSubclass, variableByteSubclass*

- In Visualworks Release 5i werden Klassen durch Senden einer Botschaft an einen *Namespace* definiert.

Methodenprotokoll *Namespace>>definition*

```
defineClass: className superclass: superID indexedType: typeName private: isPrivate
    instanceVariableNames: iVars classInstanceVariableNames: ciVars
    imports: pools category: category
```

5.9. Grundstruktur einer Methode in Smalltalk

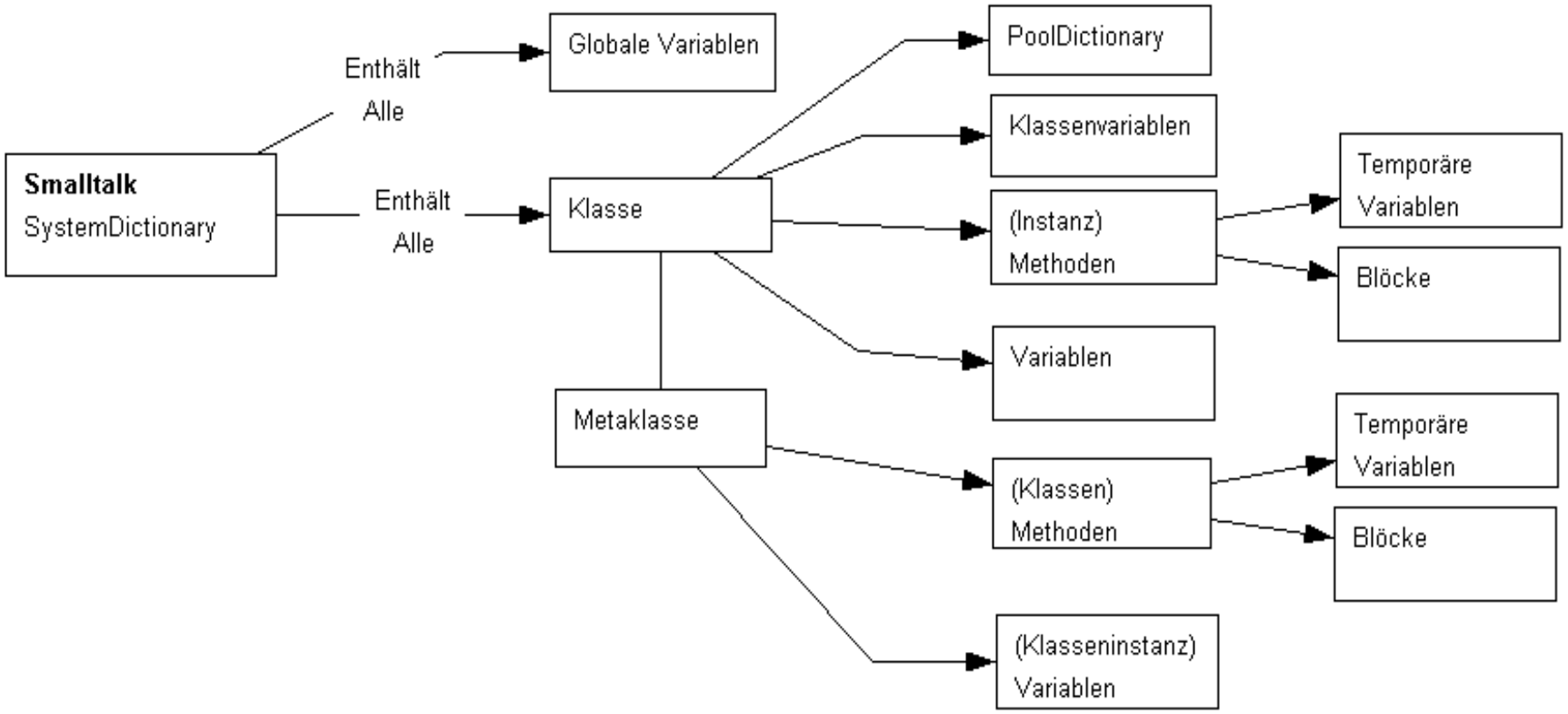
message selector and argument names
    "comment stating purpose of message"
    | temporary variable names |
    statements

Beispiel:
categories
    "This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
    ^categories isNil
        ifTrue:
            [categories := SelectionInList new]
        ifFalse:
            [categories]
```

5.10. Sichtbarkeit - Wer kennt wenn?

(Smalltalk-80 und VisualWorks bis Release 3.0)



5.11. Schreibkonventionen

Großschreibung

- Klasse, Klassenvariablen

(empfohlen)

- Klassenkategorie, globale Variable

Kleinschreibung

- Methoden, Instanzvariablen, temporäre Variablen, Klasseninstanzvariablen

Zusammengesetzte Begriffe

Die einzelnen Teilbegriffe werden groß geschrieben.

- WindowManager

Keine Bindestriche

- Window_Manager

6. Smalltalk Klassenbibliothek

etwas Statistik

Klassen im Smalltalk-System

```
Smalltalk allClasses size. 1741
```

Definierte Instanzmethoden

```
|count|
count := 0.
Smalltalk allClasses do: [ :c | count := count + c selectors size].
count. 29233
```

Definierte Instanzbotschaften

```
|coll|
coll := Set new.
Smalltalk allClasses do: [ :c | coll addAll: c selectors ].
coll size. 15082
```

Definierte Klassenmethoden

```
|count|
count := 0. Smalltalk allClasses do: [ :c | count := count + c class selectors size].
count. 6257
```

Definierte Klassenbotschaften

```
|coll|
coll := Set new. Smalltalk allClasses do: [ :c | coll addAll: c class selectors ].
coll size.  4302
```

Definierte Methoden

```
|count|
count := 0. SystemUtils allBehaviorsDo: [ :c | count := count + c selectors size].
count. 35490 (29233 + 6257)
```

Definierte Botschaften

```
|coll|
coll := Set new. SystemUtils allBehaviorsDo: [ :c | coll addAll: c selectors ].
coll size. 18762        (15082 + 4302 -> 19384) ?
```

6.1. Object - Was alle Objekte verstehen

- Vergleichen (=, ==, hash)
- Kopieren (copy, shallowCopy, postCopy)
- Fehlerbehandlung (doesNotUnderstand, halt, error:)
- Botschaften ausführen (perform:)
- Class memberschip (class, respondsTo:, isKindOf:)
- Inspektion (inspect, browse, allOwners)
- Abhängigkeitsprotokoll (changing, updating, dependents access)
  (siehe MVC, Abhängigkeitsmechanismus)
- Testen (isNil, isBehavior)

6.2. Behavior-Hierarchie

```
Object ()
    Behavior ('superclass' 'methodDict' 'format' 'subclasses')
        ClassDescription ('instanceVariables' 'organization')
            Class ('name' 'classPool' 'sharedPools')
            Metaclass ('thisClass')
```

- Zugriff auf Definition der Klasse (Name, Methoden, Variablen)
- Klassenhierarchie (superclass(es), subclasses, Aufbau der Hierachie)
- Instanzen (Erzeugung, allInstances)
- Compilieren von Methoden

6.3. UndefinedObject

- Klasse des speziellen Objekts **nil.**
- nil ist einzige Instanz der Klasse UndefinedObject.
- nil wird defaultmäßig als Nullobjekt verwendet.
- Tests: isNil, notNil
- Klasse *UndefinedObject* redefiniert Methoden von *Object*, damit *nil* in den meisten Fällen wie ein Objekt behandelt werden kann.

6.4. Boolean

Boolean
    True
    False

true - einzige Instanz der Klasse True.
false - einzige Instanz der Klasse False.

- logische Operatoren
  &,xor:,|, not

- verkürzte Auswertung
  and:, or:

- Alternativen
  ifTrue:ifFalse:

6.5. Zeichen, Zeichenketten

Character

Object ()
    Magnitude ()
        Character ()

- normale Buchstaben: $a , $A , $1 ,
- Sonderzeichen: Character cr, Character space
- Methoden zum Vergleichen, Testen (isLowercase, isDigit), Transformieren (asUppercase, asString)

String

- ’ das ist ein String’
- Methoden zum Finden von Teilstrings mit wildcards (match:)

Symbol

- #Smalltalk
- #’Auch dies ist ein Symbol’
- Literal (Bezeichner, eindeutige Schlüssel)

```
Object ()
    Collection ()
        SequenceableCollection ()
            ArrayedCollection ()
                CharacterArray ()
                    String ()
                        ByteEncodedString ()
                            ByteString ()
                            ISO8859L1String ()
                            MacString ()
                            OS2String ()
                        GapString ('string' 'gapStart' 'gapSize')
                        Symbol ()
                            ByteSymbol ()
                            TwoByteSymbol ()
                        TwoByteString ()
```

String und Symbol verstehen das Collection-Protokoll

- ➡ Kopieren, Ersetzen, Suchen von Teilfolgen
- ➡ Iterationsmethoden
- ➡ Streams

6.6 Zahlen

```
Object ()
    Magnitude ()
        ArithmeticValue ()
            Number ()
                FixedPoint ('numerator' 'denominator' 'scale')
                Fraction ('numerator' 'denominator')
                Integer ()
                    LargeInteger ()
                        LargeNegativeInteger ()
                        LargePositiveInteger ()
                    SmallInteger ()
                LimitedPrecisionReal ()
                    Double ()
                    Float ()
            Point ('x' 'y')
```

- Grundoperationen, Vergleiche
- Integerfunktionen (factorial, gcd, div, mod)
- Integer (For-Schleifen)
- mathematische Funktionen (cos, sin, tan, arcsin, arccos, arctan, sqrt, ln, exp)

Darstellung

- führende Zahl vor Dezimalpunkt
  0.001
- Exponentialschreibweise
  1.234e3 -> 1234.0
- Andere Basis (z.B. Hex-Zahl)
  11rAA -> 120
  16rAF -> 175
  keine Dezimalzahlen
- Double
  5.5 asDouble -> 5.5d
- Fraction
  5/6, 5/6 asFloat

6.7. Collectionen

Array

- #(4 'dfg' 5.6) Elemente dürfen nur Literale sein !

- #(Person new 4 5) #(#Person #new 4 5) Interpretation als Symbole

Alternative:

- Array withArguments:aCollection

- aCollection asArray

☛ Arrays haben in Smalltalk nicht so eine große Bedeutung wie z.B. in C oder Pascal, da es eine Vielzahl von weiteren Collections gibt.

Object ()
    Collection ()
        Bag ('contents')
        KeyedCollection ()
        SequenceableCollection ()
            ArrayedCollection ()
                Array ()
                    Stack ('topPtr')
                CharacterArray ()
                    String ()
                        Symbol ()
                    Text ('string' 'runs')
                List ('dependents' 'collection' 'limit' 'collectionSize')
                    DependentList ()
                WeakArray ('dependents')
            Interval ('start' 'stop' 'step')
            LinkedList ('firstLink' 'lastLink')
            OrderedCollection ('firstIndex' 'lastIndex')
                SortedCollection ('sortBlock')
        Set ('tally')
            Dictionary ()
                IdentityDictionary ('valueArray')
            IdentitySet ()

Collection

- dynamisch
- heterogen
- unterstützt eine Vielzahl von Operationen
  - Einfügen, Entfernen
  - Testen
  - Kopieren
  - Aufzählen, Iterieren

Unterklassen

- geordnet
- sortiert
- indiziert
- eindeutig

6.8. Streams

„stream over a collection“

- merkt sich die aktuelle Position innerhalb der Collection.
- Aufbau von Collections, Suchen in Collections, Iterieren über Collections
- sequentieller Zugriff

einheitlicher Zugriff auf:

- externe Quellen (Files, Datenbanken)
- Strings, Text
- Collections

```
Object ()
    Stream ()
        PeekableStream ()
            PositionableStream ('collection' 'position' 'readLimit' 'writeLimit' 'policy')
                ExternalStream ()
                        ExternalReadStream ()
                            CodeReaderStream ('swap' 'isBigEndianPlatform' 'scratchBuffer')
                        ExternalWriteStream ()
                            CodeWriterStream ('scratchBuffer')
                InternalStream ()
                    ReadStream ()
                    WriteStream ()
                        ReadWriteStream ()
                        TextStream ('lengths' 'emphases' 'currentEmphasis' 'runStartPosition')
        Random ('seed' 'increment' 'modulus' 'fmodulus' 'multiplier')
```

7. Das Model-View-Controller Framework

7.1. Abhängigkeitsmechanismus

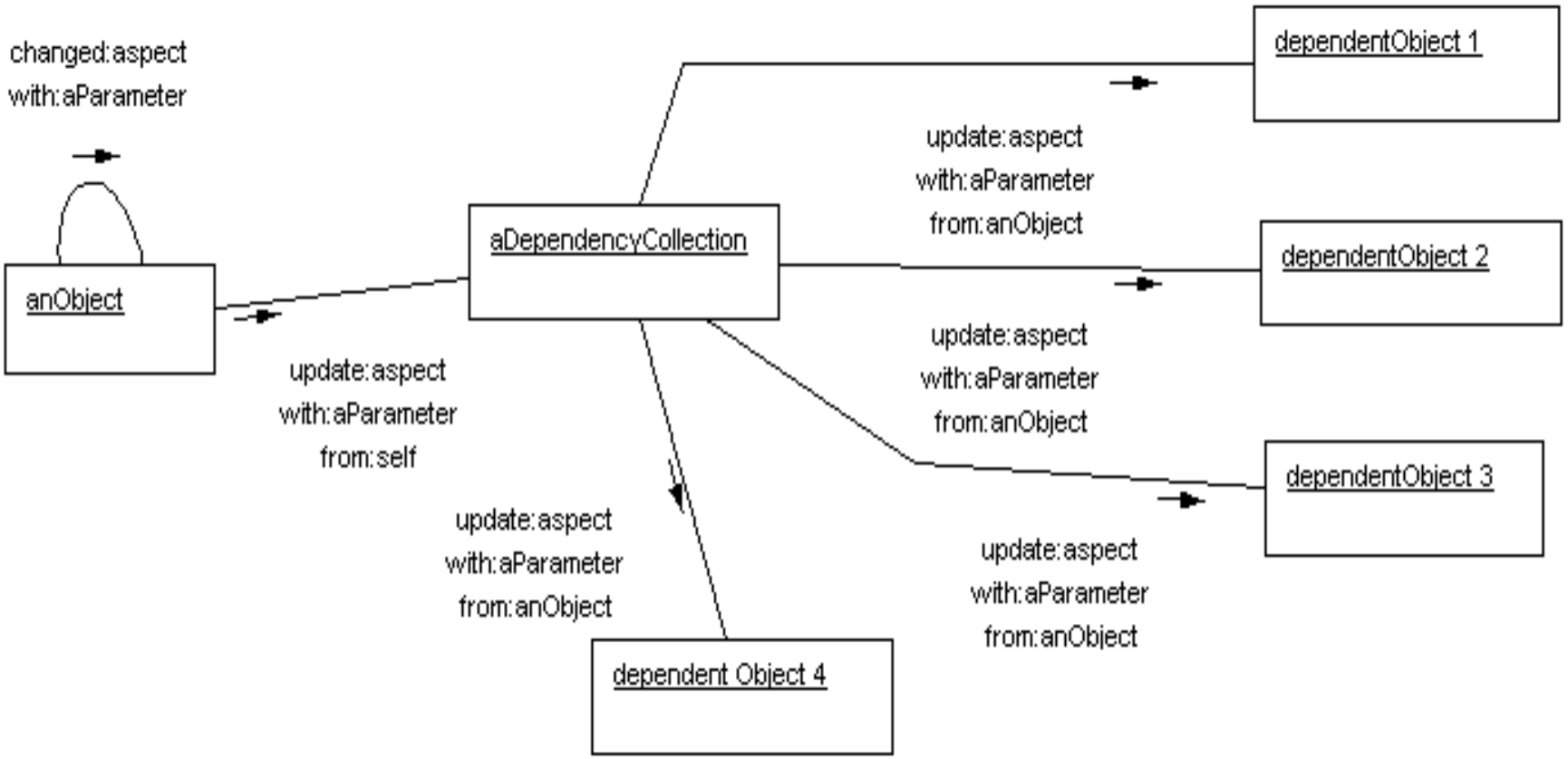
Mit Hilfe dieses Mechanismus können beliebig viele Objekte von einem anderen Objekt abhängig gemacht werden. Dadurch kann ein Objekt nach Empfang einer Nachricht, die für die abhängigen Objekte Konsequenzen hat, diese Objekte entsprechend benachrichtigen. Er ist durch das ***Changed-Update-Protokoll*** in der Klasse ***Object*** realisiert.

Arbeitsweise

1. Das geänderte Objekt schickt eine ***Changed***-Botschaft an sich selbst. Um eine bessere Kennzeichnung der geschehenen Änderung zu erreichen, kann man die Changed-Botschaft auch mit einem Aspekt und verschiedenen Parametern versehen.

2. Diese Changed-Botschaft bewirkt, daß das Objekt nun eine ***Update***-Botschaft an jedes Objekt schickt, das als abhängig markiert ist.

3. Die zugehörige ***Update***-Methode des abhängigen Objektes führt dann eine Update-Operation aus, falls der Aspekt dies signalisiert.


changed:aspect
with:aParameter
anObject
aDependencyCollection
dependentObject 1
dependentObject 2
dependentObject 3
dependent Object 4
update:aspect
with:aParameter
from:self
update:aspect
with:aParameter
from:anObject
update:aspect
with:aParameter
from:anObject
update:aspect
with:aParameter
from:anObject
update:aspect
with:aParameter
from:anObject

Realisierung einer Abhängigkeitsbeziehung

Um ein Objekt von einem anderen Objekt abhängig zu machen:
addDependent:

Um die Abhängigkeitsbeziehung aufzulösen:
removeDependent:

Nur während des Aufbaus und der Freigabe der Abhängigkeitsbeziehung muß eines der beiden Objekte Zugriff auf das andere Objekt haben. Ansonsten sind beide unabhängig voneinander.

Vorteile des Abhängigkeitsmechanismus

Ein Objekt muß nicht wissen, welche und wieviele Objekte von ihm abhängig sind.

Es schickt einfach eine Changed-Botschaft, und das weitere Vorgehen ist Aufgabe der abhängigen Objekte. Insbesondere wissen diese, welche Zustandsänderungen für sie interessant sind.

Warnung:

Dieser Mechanismus ist sehr mächtig und flexibel, birgt aber auch die Gefahr, daß bei zu intensiver Nutzung die Systemstruktur schwer verständlich wird.

Beispiel: Gummibandlinien in einem graphischen Editor

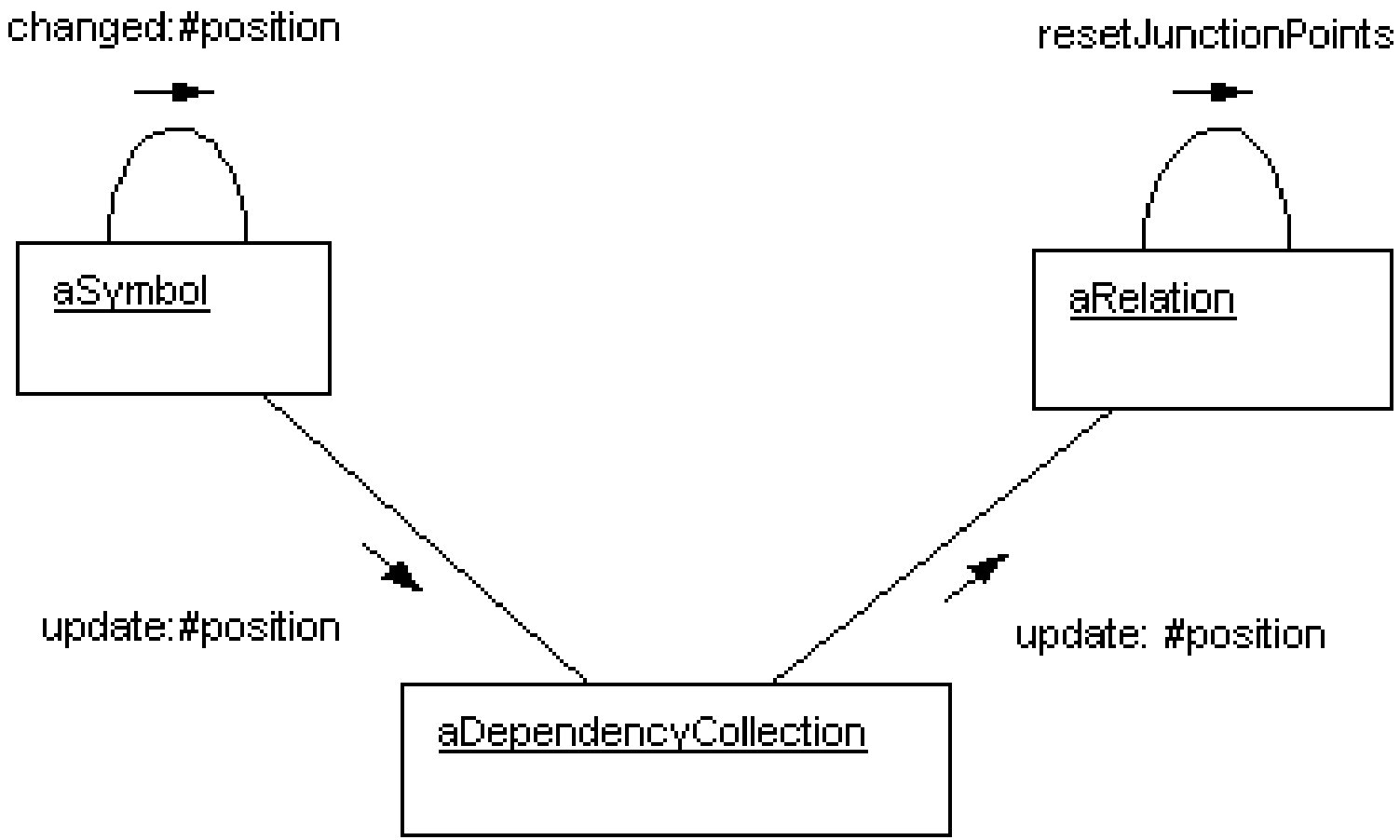
Problem

Ein Symbol ändert seine Position. Auch Änderung der Verbindungspunkte der zugehörigen Beziehungen

Lösung

Der Abhängigkeitsmechanismus bietet hier eine einfache Lösung an, ohne daß das Symbol die Anzahl oder die Art der Beziehungen kennen muß.

Das Symbol sendet eine Changed-Botschaft mit dem Aspekt #position an sich, und der Abhängigkeitsmechanismus übernimmt die Aufgabe, alle abhängigen Objekte zu informieren. Die abhängigen Objekte führen nun ihre Update-Methoden aus, in denen die Reaktion auf die Positionsänderung implementiert ist. Bei der Beziehung erfolgt nun die Aktualisierung der Verknüpfungspunkte.

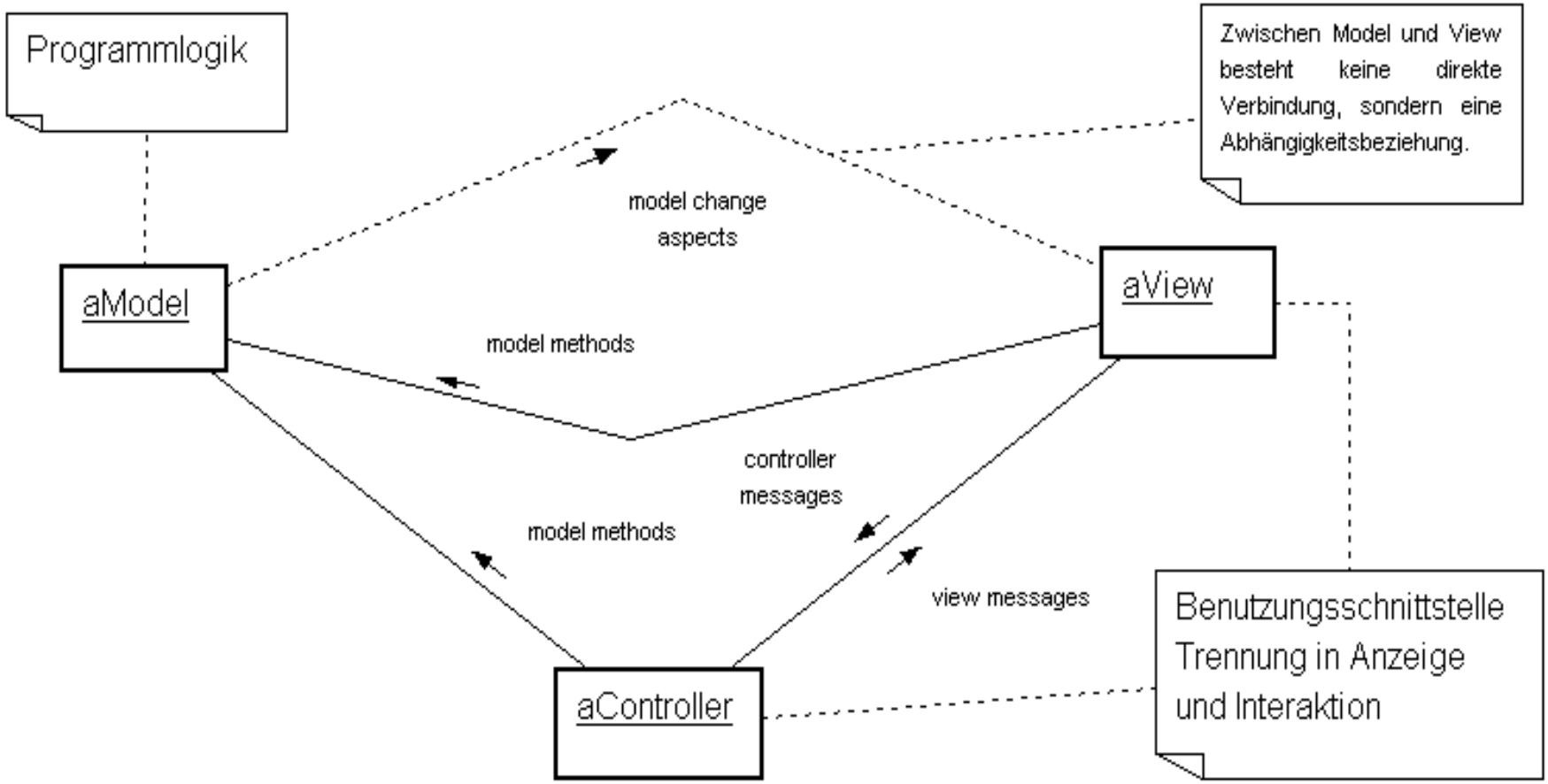

changed:#position
resetJunctionPoints
aSymbol
aRelation
update:#position
update: #position
aDependencyCollection

7.2. Das Model-View-Controller Paradigma

Ein zentrales Entwurfsprinzip in der Softwareentwicklung:

Trennung von Programmlogik und Benutzungsschnittstelle

- völlig separate Entwicklung der problemrelevanten Strukturen.
- Verbinden der Programmlogik mit verschiedenen Benutzungsschnittstellen, ohne daß an der Logik etwas geändert werden muß.


Programmlogik
Zwischen Model und View besteht keine direkte Verbindung, sondern eine Abhängigkeitsbeziehung.
model change aspects
aModel
aView
model methods
controller messages
model methods
view messages
aController
Benutzungsschnittstelle
Trennung in Anzeige und Interaktion

Umsetzung des MVC-Paradigmas in Smalltalk

- Implementation des MVC-Paradigmas mittels des Abhängigkeitsmechanismus.

- Trennung der Benutzungsschnittstelle in zwei Teile, den View und den Controller.
  Der **Controller** ist für die Benutzereingaben verantwortlich.
  Der **View**-Teil übernimmt die Darstellung des Modells auf dem Bildschirm.

- Das **Modell** hat keinen direkten Zugriff auf die Benutzungsschnittstelle, sondern ist nur über eine Abhängigkeitsbeziehung mit dem View verbunden.
  ➡ Binden beliebig vieler und vor allem auch verschiedener Benutzungsschnittstellen an ein Modell

- Die Benutzungsschnittstelle hat direkten Zugriff auf das Modell. Dies geschieht durch die Instanzvariable ***model*** in den Klassen ***DependentPart*** und ***Controller***.

Zusammengesetzte Views

CompositePart: Container für View-Elemente

☛ Composite-Muster

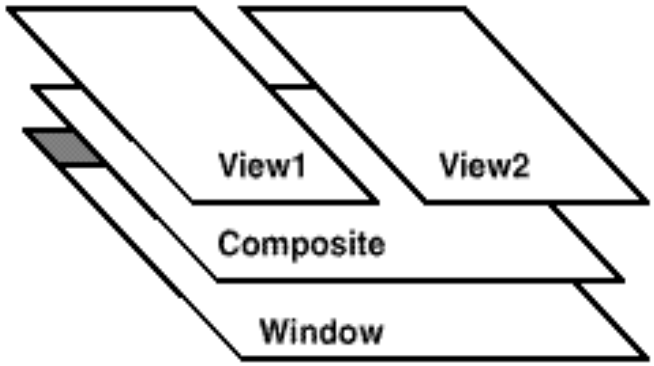


Figure 18-18 A CompositePart holds a collection of other visual components.

(VisualWorks User Guide Version 2.5)

Komplexe Modelle

- Ummanteln der Teilmodelle
- Koordination

7.3. Erweitertes Model-View-Controller Paradigma

- Vielfach besteht das Informationsmodell nicht aus einem Objekt, sondern aus einer Gruppe von Objekten.
  - ➡ Zusammenfassung mehrerer Teilaspekte verschiedener Objekte in einer Ansicht

Aber:

Jeder View kann nur mit einem Model verknüpft werden.

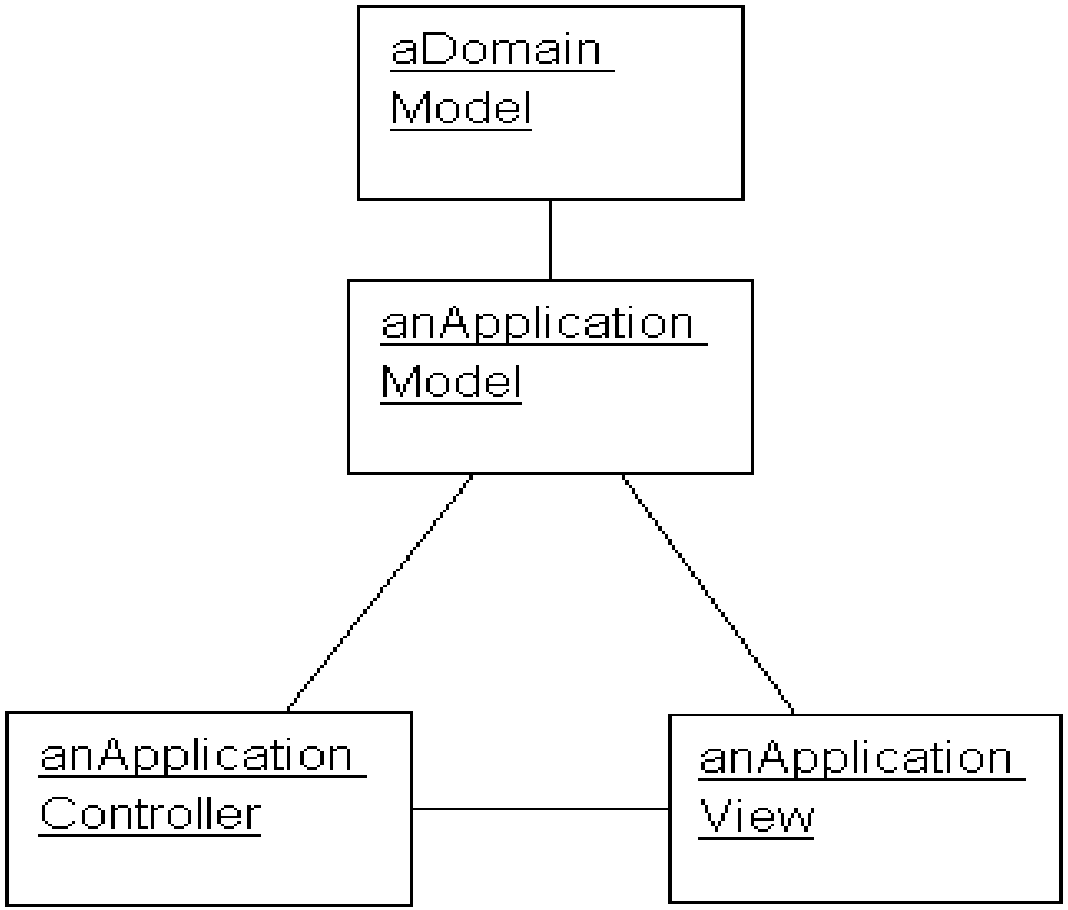
Lösung:

sogenannte ***Umbrella-Modelle***

- ☛ Gehört mehr zur Benutzungsschnittstelle als zum Informationsmodell.
- ☛ Stellt das Methodenprotokoll für die Verbindung zwischen den einzelnen Model-Objekten und dem View bereit.

ApplicationModel = Umbrella-Model
DomainModel = die einzelnen Model-Objekte

- ☛ Grundlage des Application Framework von VisualWorks


aDomain Model
anApplication Model
anApplication Controller
anApplication View

7.4. View und Wrapper

Smalltalk-80: komplexe View-Klassen
VisualWorks: Wrapperkonzept (Decorator-Pattern) und einfache View-Klassen

Squeak 2.0 (Beispiel für ursprüngliche Smalltalk-80 Implementierung)

```
Object subclass: #View
    instanceVariableNames: 'model controller superView subViews transformation viewport
                            window displayTransformation insetDisplayBox borderWidth
                            borderColor insideColor boundingBox '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Interface-Framework'
```

VisualWorks 1.0

```
DependentPart subclass: #View
    instanceVariableNames: 'controller (model container)'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Interface-Framework'
```

Wrapperhierarchie (unvollständig, VisualWorks Version 5i)

```
Object ()
  VisualComponent ()
    VisualPart ('container')
      Wrapper ('component')
        GeometricWrapper ()
          FillingWrapper ()
          StrokingWrapper ('lineWidth')
        GraphicsAttributesWrapper ('attributes')
        PassivityWrapper ('controlActive' 'visuallyActive' 'visible')
        ReversingWrapper ('reverse' 'offset')
          StrikeOutWrapper ()
        ScalingWrapper ('scale')
        TranslatingWrapper ('origin')
          LayoutWrapper ('layout')
            BoundedWrapper ('extent')
                BorderedWrapper ('insetDisplayBox' 'border' 'inset' 'insideColor')
              BoundingWrapper ()
                    ScrollWrapper ('dependents' 'preferredBoundsBlock')
```

Wrapperkonzept

- Aufbrechen der komplexen Views in mehrere Hüllen

- Wrapper umhüllen den jeweiligen View und fügen zusätzliche Funktionalität hinzu.
  Überschreiben von Botschaften - zuerst Wrapper-spezifisches Verhalten
  - dann Verhalten der Komponente

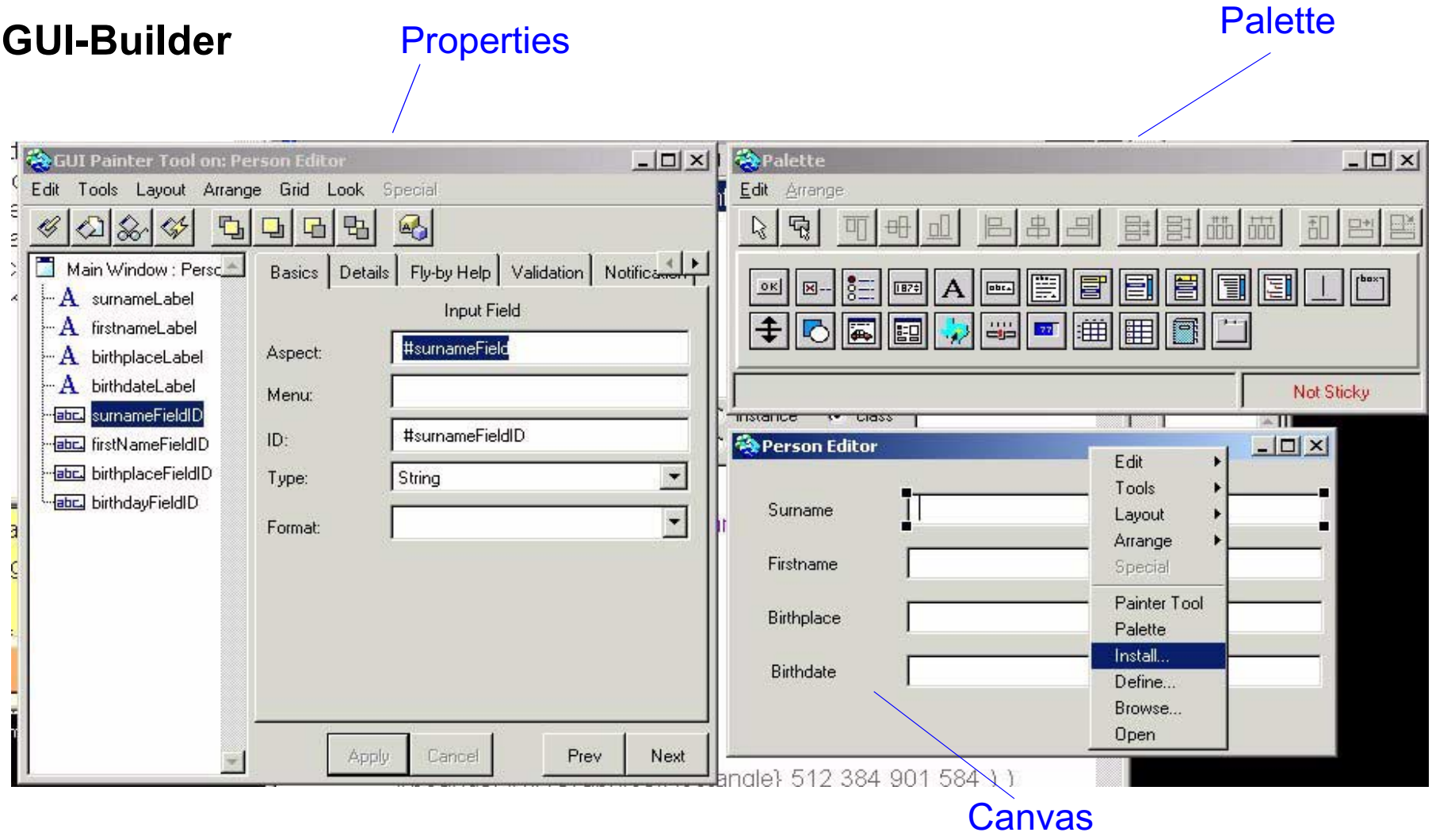
- Transparenz des Wrapperkonzepts

  Wrapper >> doesNotUnderstand: aMessage
  "The default behavior is to create a Notifier containing the
  appropriate message and to allow the user to open a Debugger.
  Subclasses can override this message in order to modify this behavior."
  "Michael Prasse 1995 .
  Falls ein Wrapper eine Botschaft nicht versteht, dann versteht sie vielleicht die component.
  Hintergrund: Die Wrapper-Struktur sollte eigentlich unsichtbar sein. Ohne diese Methode gibt es aber Probleme, Wrapper beliebig zu schachteln; zum Beispiel model-Botschaft nicht möglich, weil Wrapper diese nicht versteht"

  ^self **component** perform: aMessage selector withArguments: aMessage arguments

7.5. Application Framework von Visualworks

GUI-Builder



ApplicationModel

- abstrakte Klasse

- Grundlage für Anwendungen, die den UIBuilder für die Generierung ihrer Interfaces nutzen.

Methodenprotokoll für:
- Zusammenarbeit mit UIBuilder
- Spezifikation des Interfaces
- Verknüpfung der Modelle mit den einzelnen Widgets
- Action methods für ActionButtons
- Menüs for MenuButtons

Unterklassen:
- Bereitstellung der jeweiligen Ressource (Action, Menu, Label, Model ...) durch Definition einer Methode mit dem Namen der Ressource, die diese als value zurückgibt.

Canvas Installation

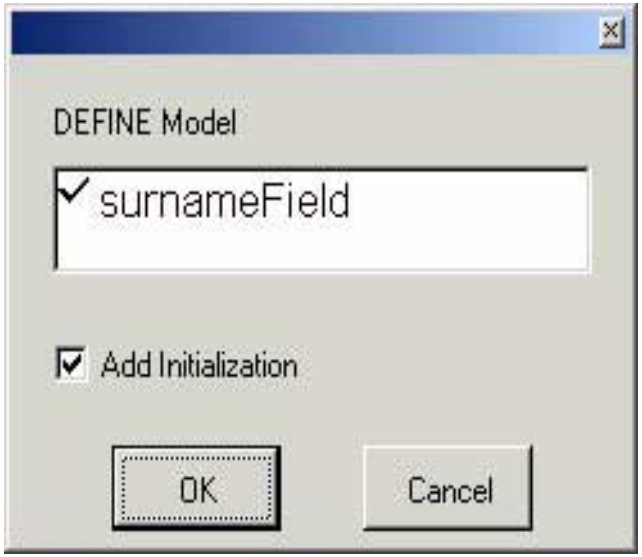

```
PersonEditorApp>>editorSpec
"UIPainter new openOnClass:
self andSelector: #editorSpec"

<resource: #canvas>
^#(#{UI.FullSpec}
#window:
#(#{UI.WindowSpec}
#label: 'Person Editor'
#bounds: #(#{Graphics.Rectangle} 512 384 901 584 ) )
#component:
#(#{UI.SpecCollection}
#collection: #(
#(#{UI.LabelSpec}
#layout: #(#{Core.Point} 21 23 )
#name: #surnameLabel
#label: 'Surname' )
#(#{UI.LabelSpec}
#layout: #(#{Core.Point} 21 60 )
... ) ) ) )
```

WindowSpecs

- Aufbau der Views über windowSpec-Methoden
  - ☛ textuelle Beschreibung, Metasprache für Fenster
  - ☛ Specs können mit GUI-Builder oder programmiertechnisch erzeugt werden
  - ☛ Speichern von Fenster-Beschreibungen (z.B. auch in Datenbank)
- spezielle UIBuilder, die aus der Spec die Views erzeugen
- Anzahl vorgefertigter Widgets

Definition von Defaults für Ressourcen-Methoden




```
PersonEditorApp>>surnameField
"This method was generated by UIDefiner.  Any edits
made here may be lost whenever methods are automa-
tically defined.  The initialization provided below may
have been preempted by an initialize method."

^surnameField isNil
ifTrue: [surnameField := String new asValue]
ifFalse:[surnameField]
```

Verwendung:

```
aPersonEditorApp surnameField value: 'Goldberg'.
```

Erzeugen und Öffnen eines Interface

```
ApplicationModel >> openInterface: aSymbol withPolicy: aPolicy inSession: anAppContext
    "Open the ApplicationModel's user interface, using the specification
    named and the given look policy and application context."

    | spec |
    builder := aPolicy newBuilder.
    builder source: self.
    spec := self class interfaceSpecFor: aSymbol.
    ...
    self preBuildWith: builder.
    ...
    builder add: spec.
    ...
    self postBuildWith: builder.
    builder openWithExtent: spec window bounds extent.
    ...
    self postOpenWith: builder.
    ^builder
```

Hook-Methoden zur Adaptierung des Erzeugungs- und Öffnen-Prozesses

Adaptierung des Erzeugungsprozesses

preBuildWith: aBuilder

"This message is sent by the builder prior to beginning construction of either a SubCanvas or a complete window."

- Anpassung des Builders vor dem Konstruktionsprozeß
- Bereitstellen von zusätzlichen Ressourcen für den Builder, die für die Konstruktion erforderlich sind, aber durch die Spec nicht bereitgestellt werden.

postBuildWith: aBuilder

"This message is sent by the builder when it has completed work on either a complete window or a SubCanvas."

- Anpassung des Interface vor dem Öffnen
- Änderungen an Komponenten
- Enable/disable von Komponenten
- Definition zusätzlicher Ereignisse und Äbhängigkeiten (onChangeSend:to:)
- Änderung des Fenstersymbols, -titels

postOpenWith: aBuilder

"This message is sent by the builder after it has opened a completed window."

- Änderungen mit Zugriff auf Fensterressourcen
- Window events

ValueModel

- *ValueModel-Framework* als Grundlage für Model im GUI-Framework

- Zwischenschalten eines ValueModel zwischen View und dargestelltem Sachverhalt
  - ☛ Mediator-Pattern
  - ☛ Vereinfachung derAbhängigkeitsbeziehungen

- einfaches einheitliches Model mit standardisierten Methodenprotokoll (*value*, *value:*)

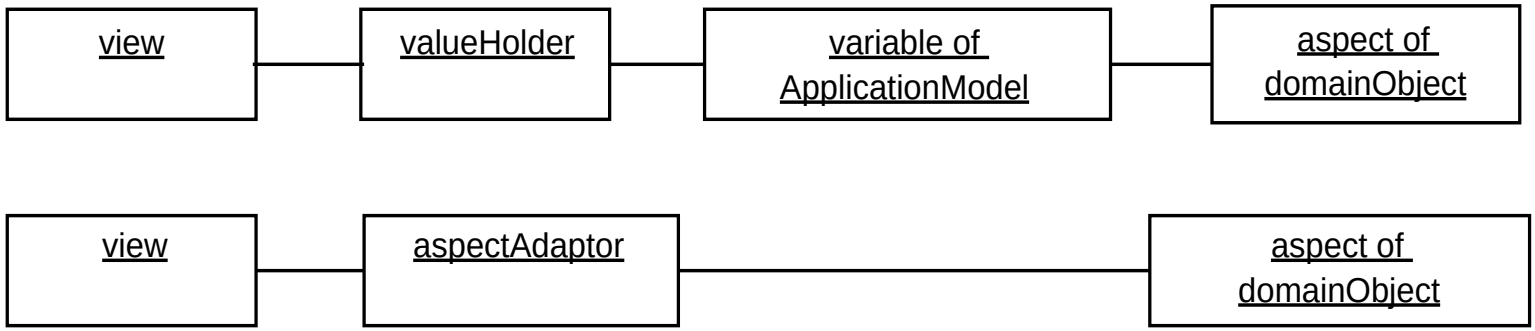
- dargestellter Sachverhalt
  - eine Variable des Applicationmodel (ValueHolder)
  - ein Aspekt eines betrachteten Domainobjekts (AspektAdaptor)

ValueHolder Hierarchie

Object ()
  Model ('dependents')

    ValueModel ()
      ComputedValue ('cachedValue' 'eagerEvaluation')
        BlockValue ('block' 'arguments' 'numArgs')
      PluggableAdaptor ('model' 'getBlock' 'putBlock' 'updateBlock')
        TypeConverter ()
      ProtocolAdaptor ('subject' 'subjectSendsUpdates' 'subjectChannel' 'accessPath')
        AspectAdaptor ('getSelector' 'putSelector' 'aspect')
        IndexedAdaptor ('index')
          SlotAdaptor ()
      RangeAdaptor ('subject' 'rangeStart' 'rangeStop' 'grid')
      ValueHolder ('value')
        BufferedValueHolder ('subject' 'triggerChannel')
        EvaluationHolder ('object')
        Proxy ('displayString' 'fileIndex')
        TextCollector ('entryStream' 'characterLimit')

Unterschied zwischen AspectAdaptor und ValueHolder



level

^(AspectAdaptor subject: self domainModel sendsUpdates: true) forAspect: #level.

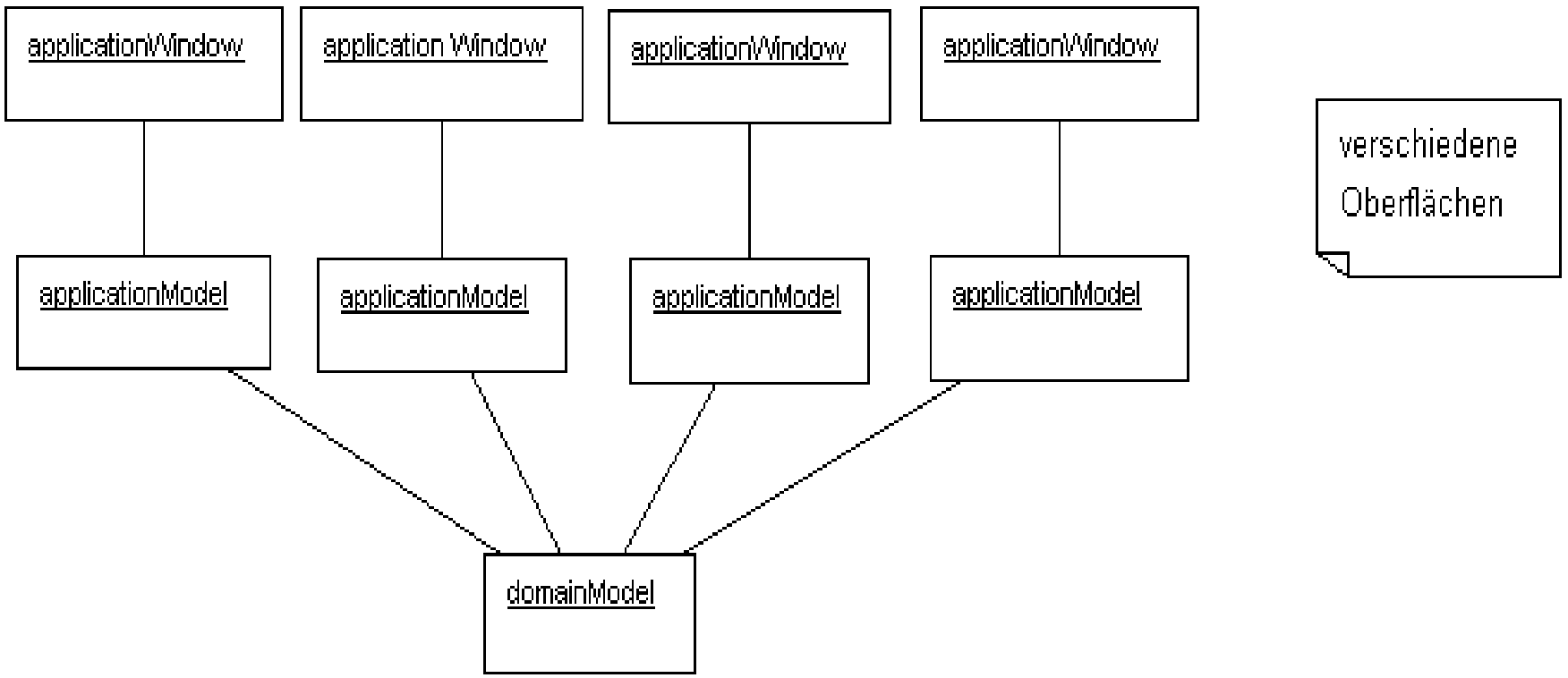
- direkte Verknüpfung und Synchronisation mit Domänenaspekt.
- Methode sollte nicht direkt aufgerufen werden, da bei jedem Aufruf Erzeugung eines neuen AspectAdaptors.

level

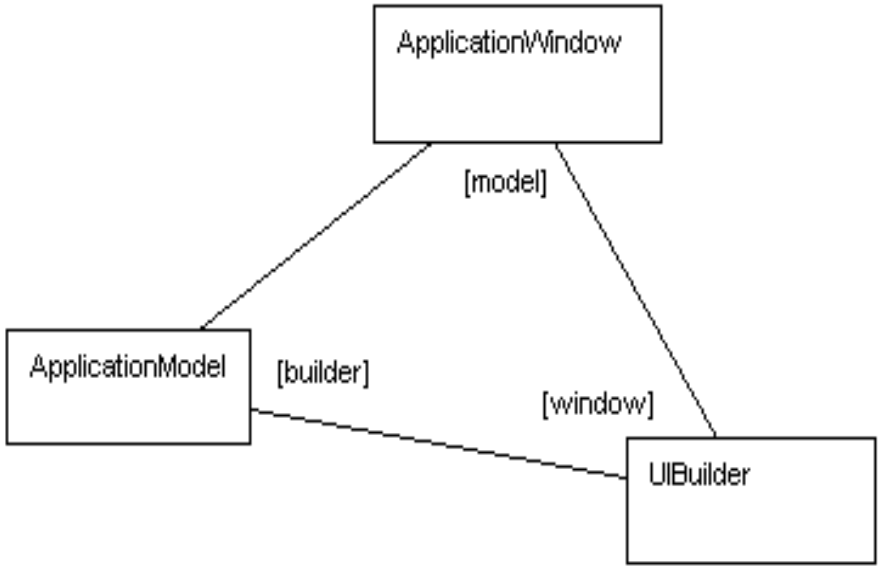
^level isNil ifTrue:[level:= Integer new asValue] ifFalse:[level]

- manuelle Verknüpfung und Synchronisation mit Domänenaspekt
- mehrmaliges Aufrufen unkritisch
- Programmierer bestimmt Zeitpunkt der Synchronisation

Verknüpfung ApplicationModel - Domäne



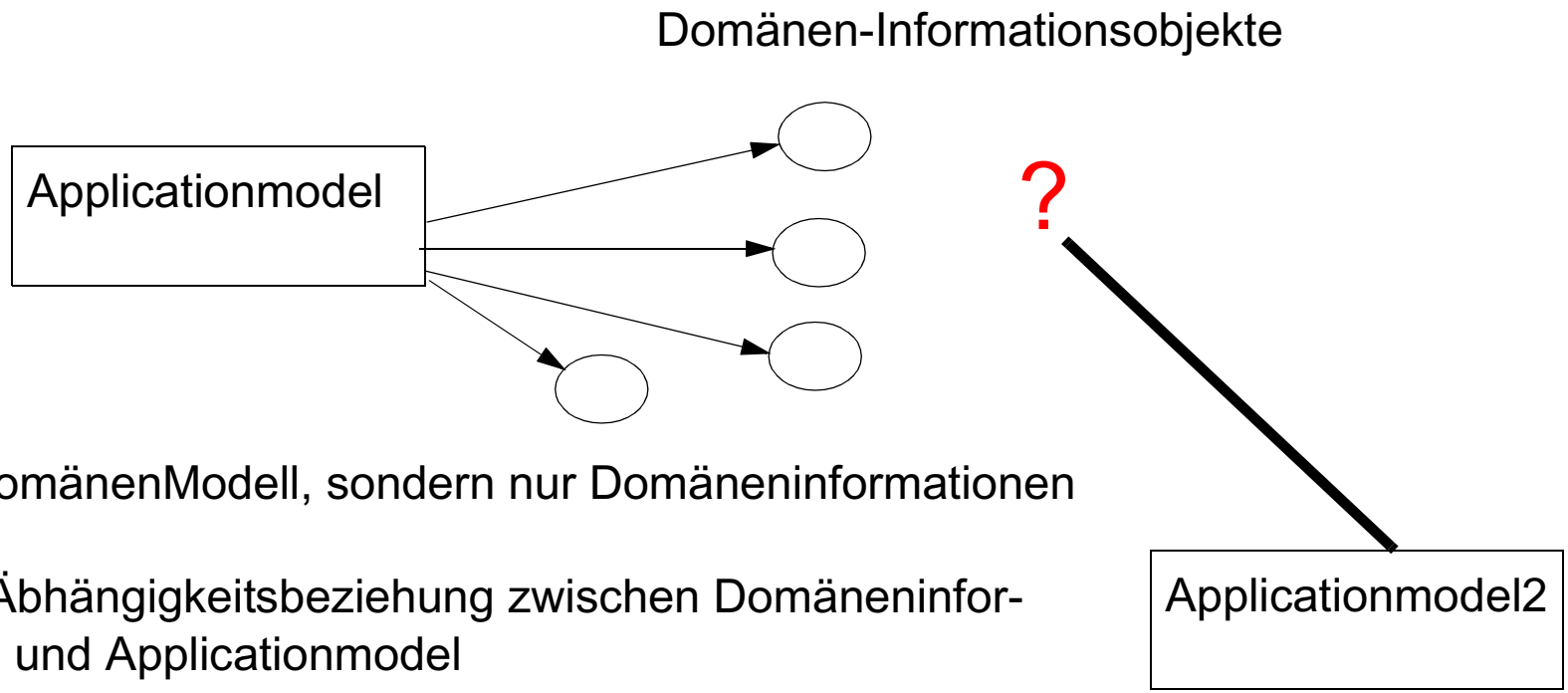
Verknüpfung ApplicationModel, ApplicationWindow und Builder



Durch den Builder hat ein ApplicationModel immer indirekt Zugriff auf sein Window. Diese Verbindung sichert, daß das ApplicationModel mit Hilfe des Builders auch während der Laufzeit das Window ändern kann.

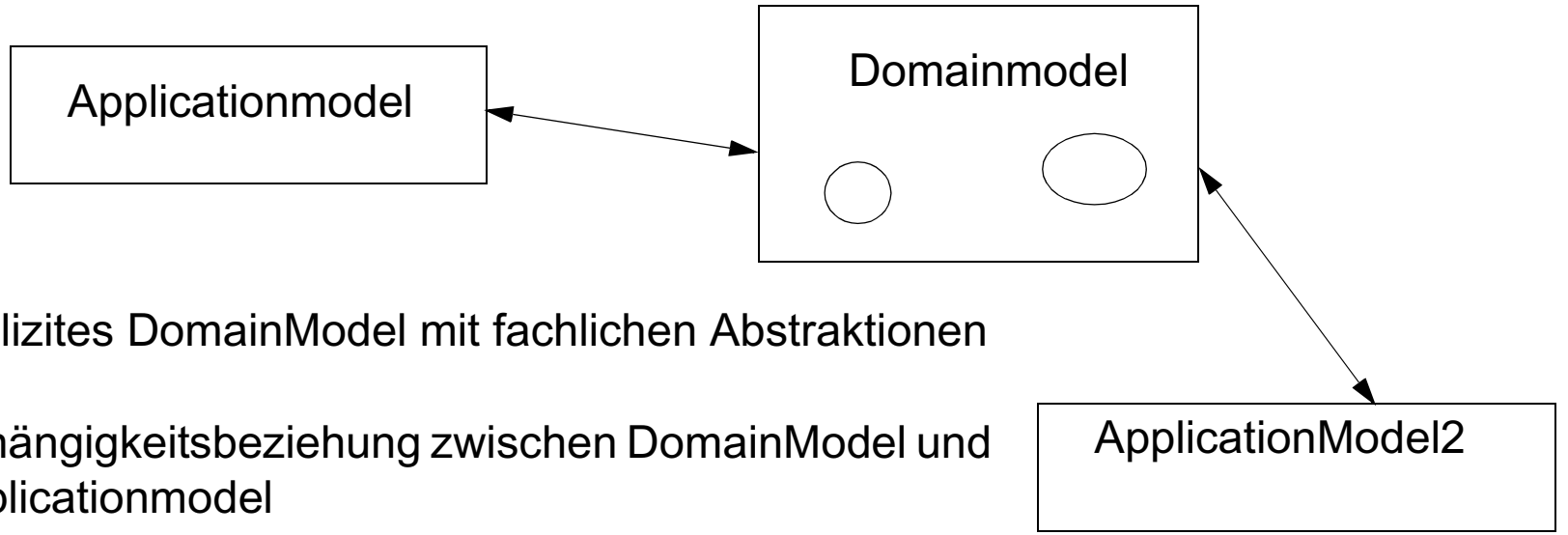
7.6. Domäne

Domänen-Informationsobjekt-Ansatz



- kein DomänenModell, sondern nur Domäneninformationen
- keine Äbhängigkeitsbeziehung zwischen Domäneninformation und Applicationmodel
- Anbindung weiterer Applikationsmodelle ?
- Applikationsmodell kapselt Teil der fachlichen Zusammenhänge

Domänenmodell-Ansatz



- explizites DomainModel mit fachlichen Abstraktionen
- Abhängigkeitsbeziehung zwischen DomainModel und Applicationmodel
- ApplicationModel gehört zum User Interface
- Ankopplung weiterer Applicationmodelle möglich
- GUI-lose Verwendung
- leichterer Austausch der GUI

7.7 Canvas und Subcanvas

Modularisierung und Wiederverwendung objektorientierter Oberflächen

- Reduzierung des Implementierungs- und Wartungsaufwandes
- Redundanzvermeidung
- Einheitlichkeit
- Anwendung objektorientierter Modellierungstechniken (Komposition) auch bei der Entwicklung von Oberflächen
- Grundlage für die Entwicklung von Oberflächen, die zur Laufzeit dynamisch ausgetauscht werden können.

Wiederverwendungsstrategien

- neues Canvas für existierendes Applikations- und Domänenmodell
- Austausch des Domänenmodells (gleiches Application-Modell und Canvas)
- Verknüpfen existierender Applikationen über Aufrufbeziehungen
- Integration eines Canvas in ein anderes
  - Copy & Paste
  - Subclassing
  - Subcanvas-Integration ohne eigenen Client
- Einbetten von Subapplikationen (Komposition)
- Vererbung von Applikationsmodellen

Weiterführende Literatur:

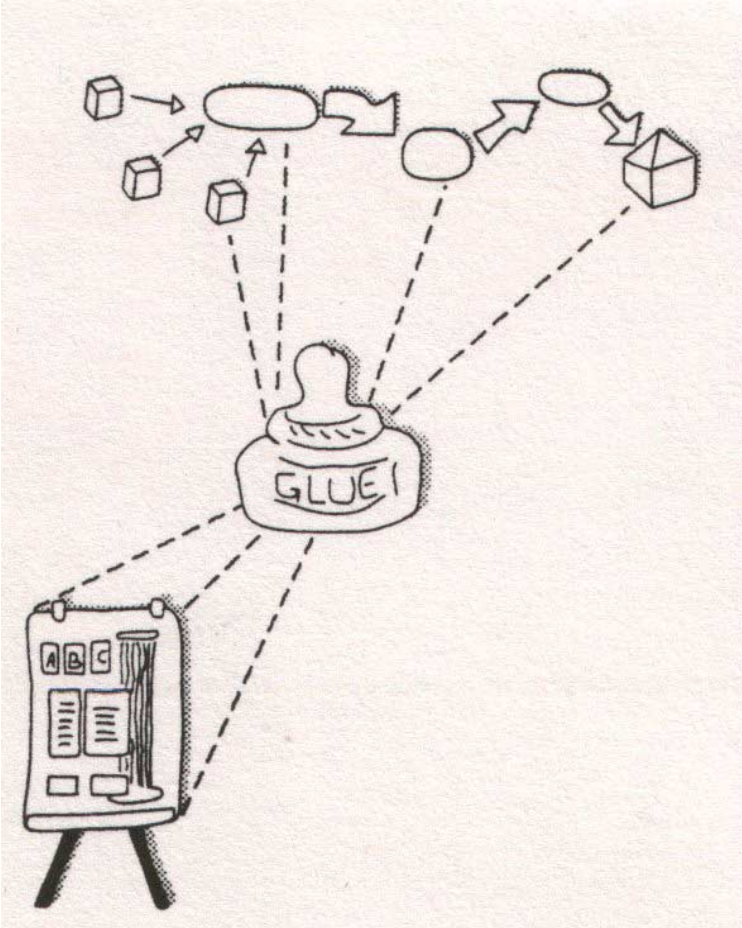
[VW 1992] *„VisualWorks Tutorial“*, VisualWorks Version 1.0., ParcPlace Systems, 1992, Kapitel 4 „Visual Reuse“ , S. 69 ff.

Grundstruktur einer Applikation

domain model

application model

canvas (user interface)



Reuse aspects:

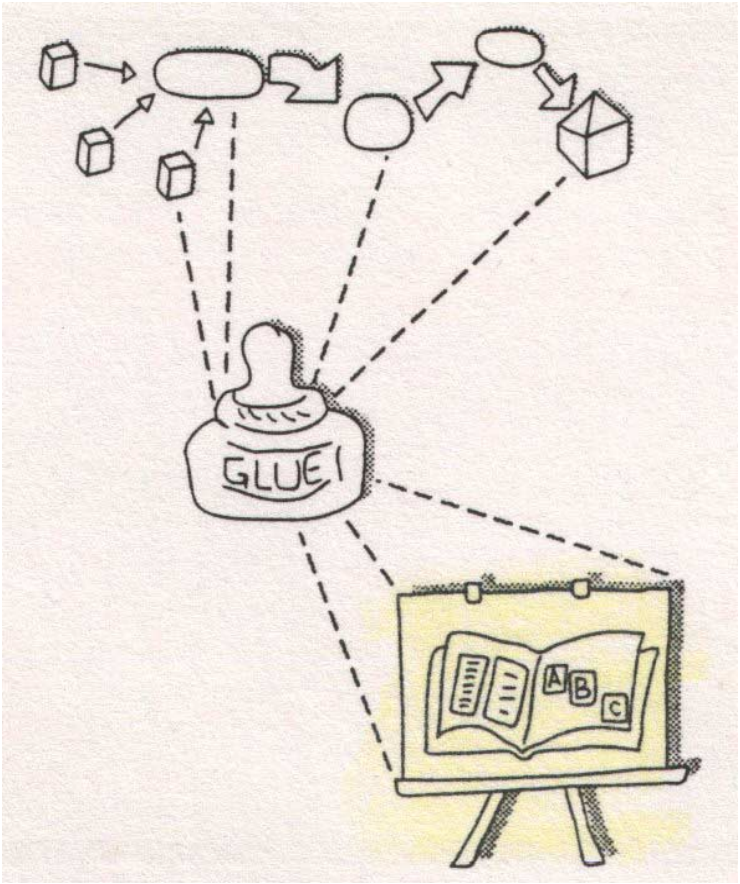
1). domain model

2). application model

3). canvas

([VW 1992], S. 70)

Neues GUI für existierende Applikations- und Domänenmodelle



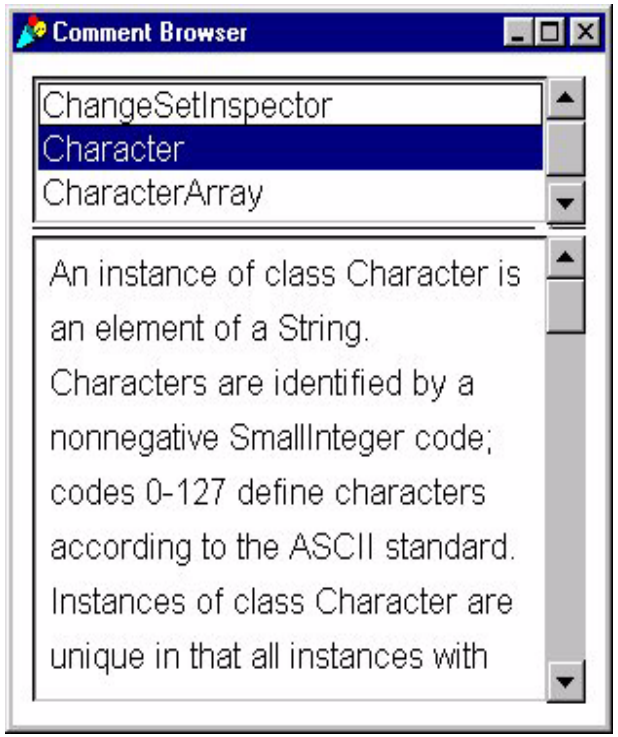
Reuse: models

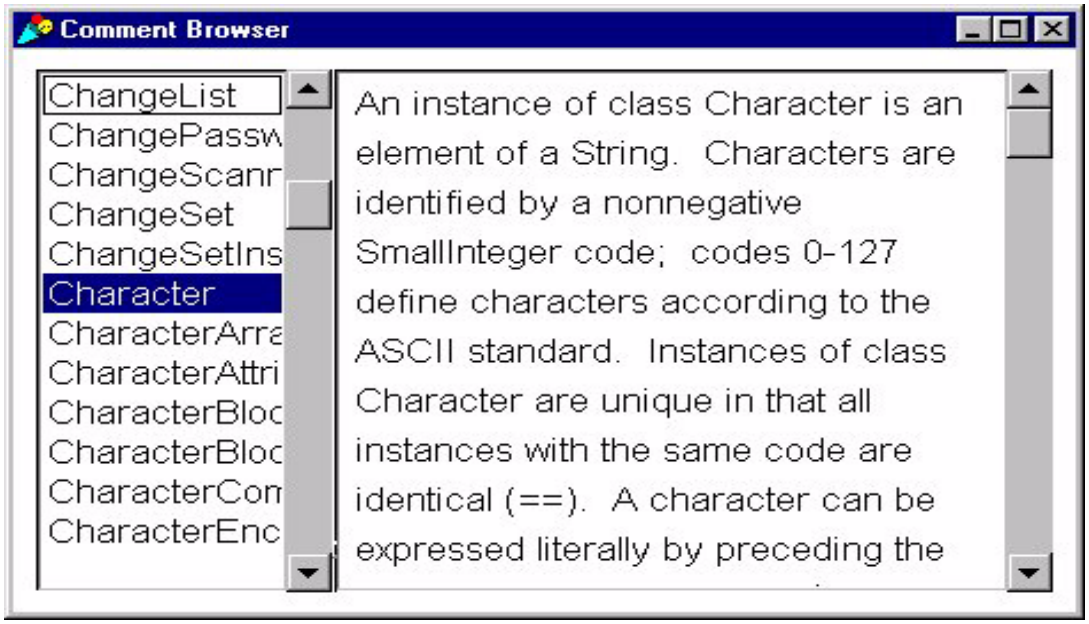
new canvas

([VW 1992], S. 70)

Beispiel: CommentBrowser (Object Reference)

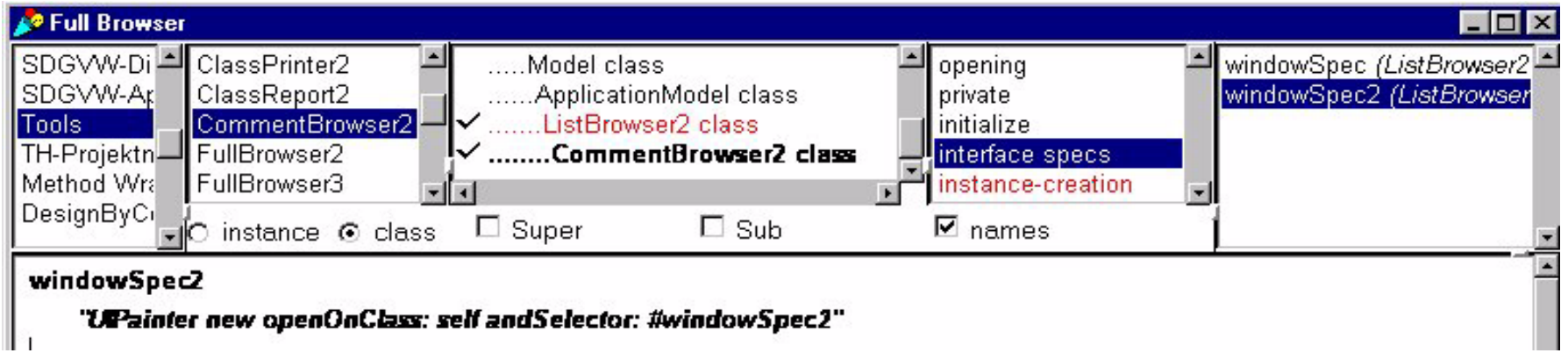
- horizontale und vertikale Anordnung





Aufrufvarianten: CommentBrowser2 open.
CommentBrowser2 openVertical.

CommentBrowser windowSpec-Methoden



- CommentBrowser2 (ListBrowser2) definiert zwei windowSpec-Methoden
- Je nach gewünschtem Fensterlayout Einsatz der entsprechenden windowSpec

openVerticalListBrowserOn: aCollection label: labelString initialSelection:sel

"Create and schedule a browser on the collection of messages aCollection. "

^self openListBrowserOn: aCollection
label: labelString
initialSelection:sel
interface: ***#windowSpec2***

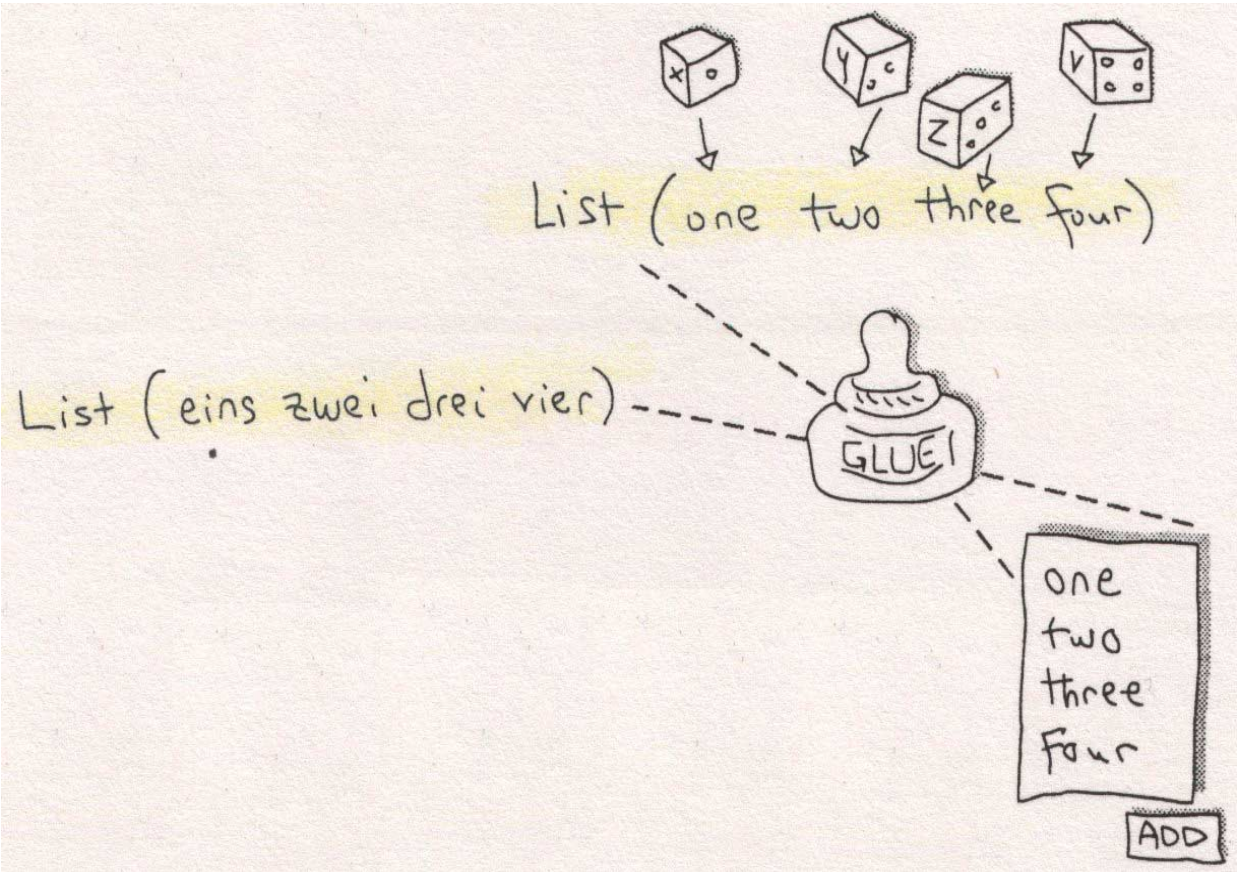
openListBrowserOn: aCollection label: labelString initialSelection:sel

"Create and schedule a browser on the collection of messages aCollection. "

^self openListBrowserOn: aCollection
label: labelString
initialSelection:sel
interface: ***#windowSpec***

Austauschbare Domänenmodelle

different domain models



Reuse: application model and canvas

([VW 1992], S. 71)

Beispiel: SimpleDialog

```
SimpleDialog new
choose: 'Wähle eine Zahl'
labels: (Array with: 'Eins' with: 'Zwei' with: 'Drei')
values: #(1 2 3)
default: 1.
```


```
SimpleDialog new
choose: 'Are you tired yet?'
labels: (Array with: 'absolutely' with: 'sort of'
        with: 'not really')
values: #(#yes #maybe #no)
default: #maybe.
```

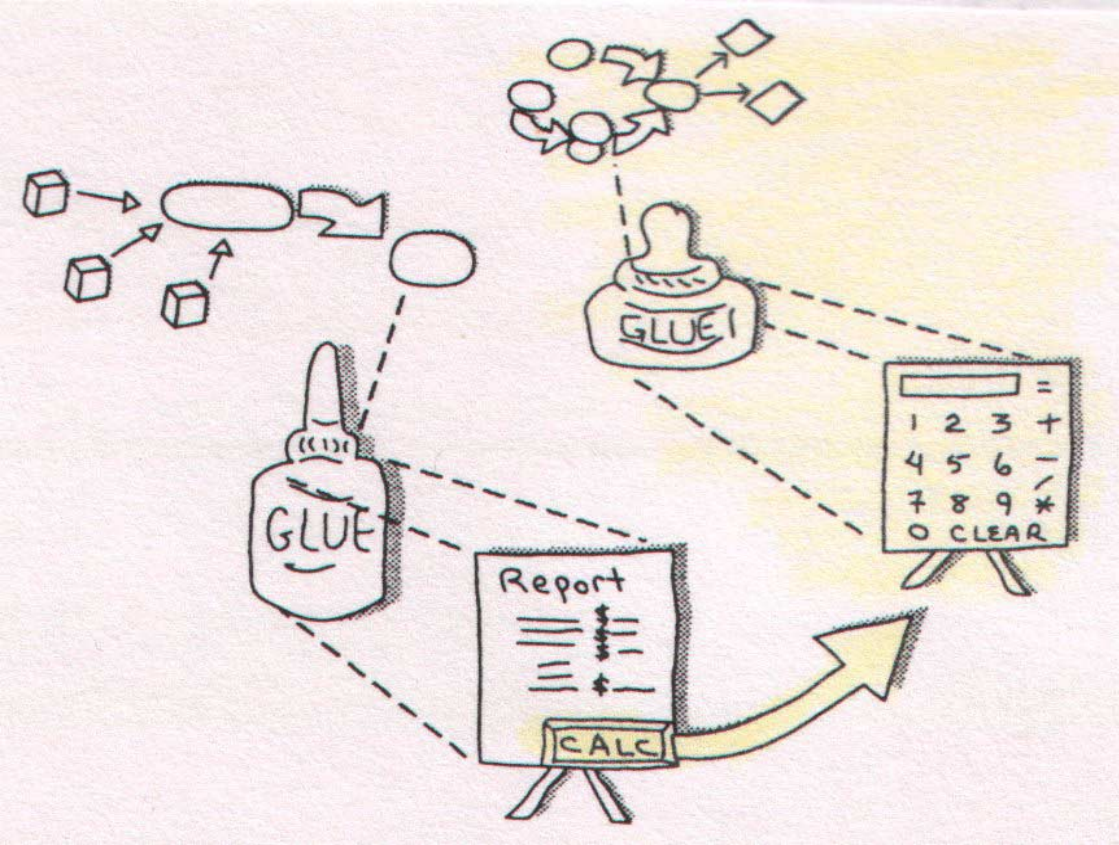
Generell:

- einfaches Austauschen des Domänenmodells
- Voraussetzung: gleiches Methodenprotokoll bezüglich Applikationsmodell
- generische Benutzungsschnittstellenkomponenten (Listen, Dialoge ...)
- Abstraktion von den konkret angezeigten Daten

☛ Trennung von Modell und Darstellung (MVC)

Verknüpfung existierender Applikationen

- Aufbau komplexer Anwendungen aus existierenden Anwendungen durch Aufrufbeziehungen

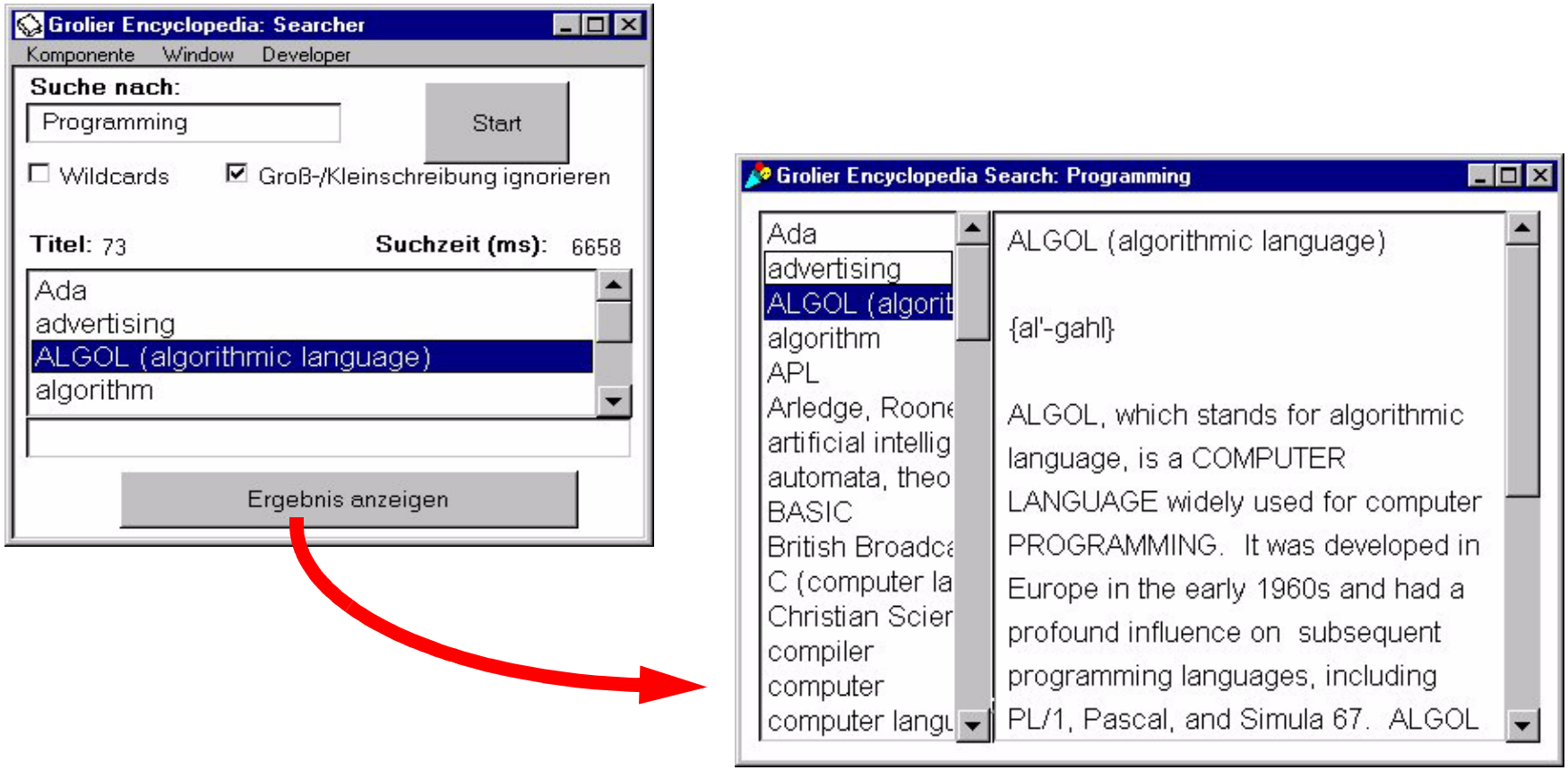


Reuse: gesamte Anwendung

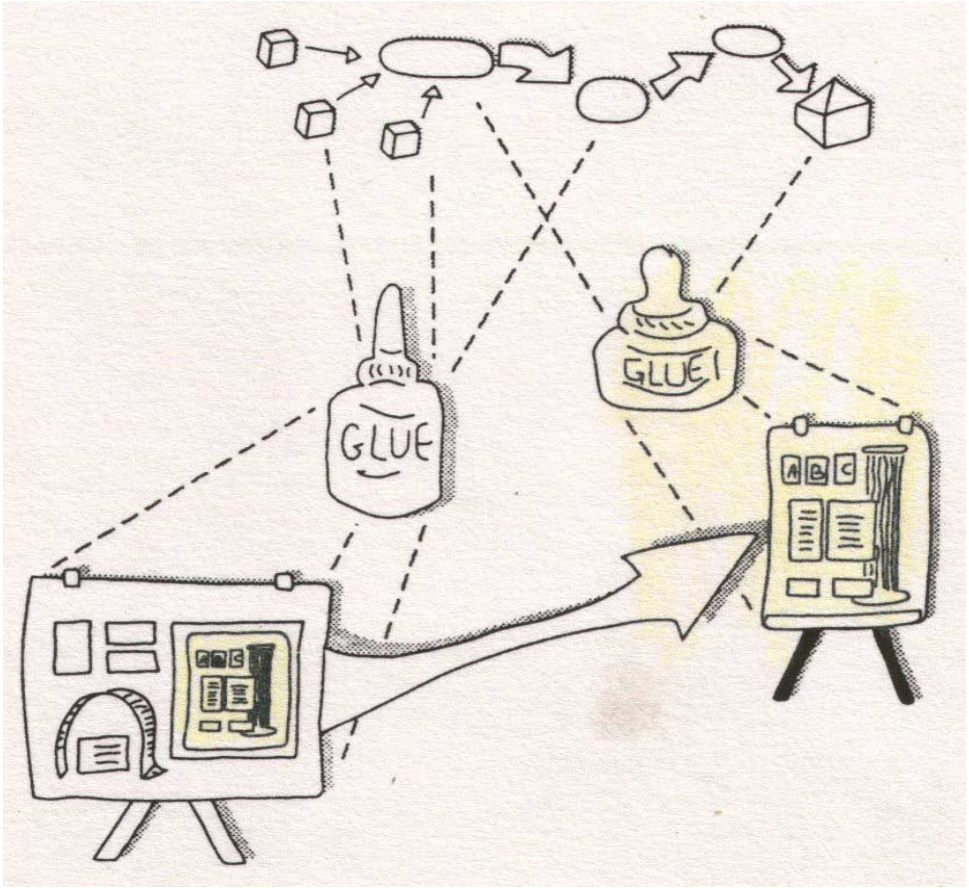
([VW 1992], S. 74)

Beispiel: Grolier Encyclopedia (Simple Text Interface)

- Verknüpfung durch Action Buttons, Menü-Einträge, Tool Bars ...
- verwendet für Anwendungen, Hinweisfenster, Eingabedialoge ...
- Multiple Windows Applications



Einbettung von Teilanwendungen



Reuse:
canvas und
subapplication
model innerhalb
des übergeordneten
application model

([VW 1992], S. 73)

Komposition von Applikationsmodellen

- wichtigste Form der Wiederverwendung eines canvas
- erlaubt Komposition von Applikationen

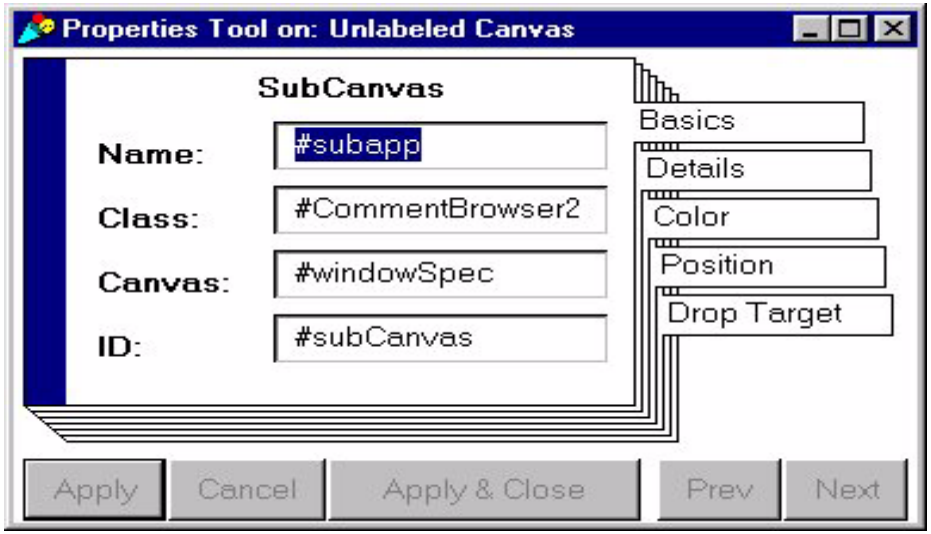
➡ Modularer Aufbau von Benutzungsschnittstellen

➡ Grundlage für die Entwicklung dynamisch änderbarer Oberflächen

Vorgehen:

- Einfügen eines subcanvas-Widget
- Festlegen des Namens
- Angabe der wiederzuverwendeten Canvas-Spec
- Angabe der zugehörigen Klasse für die Canvas-Spec

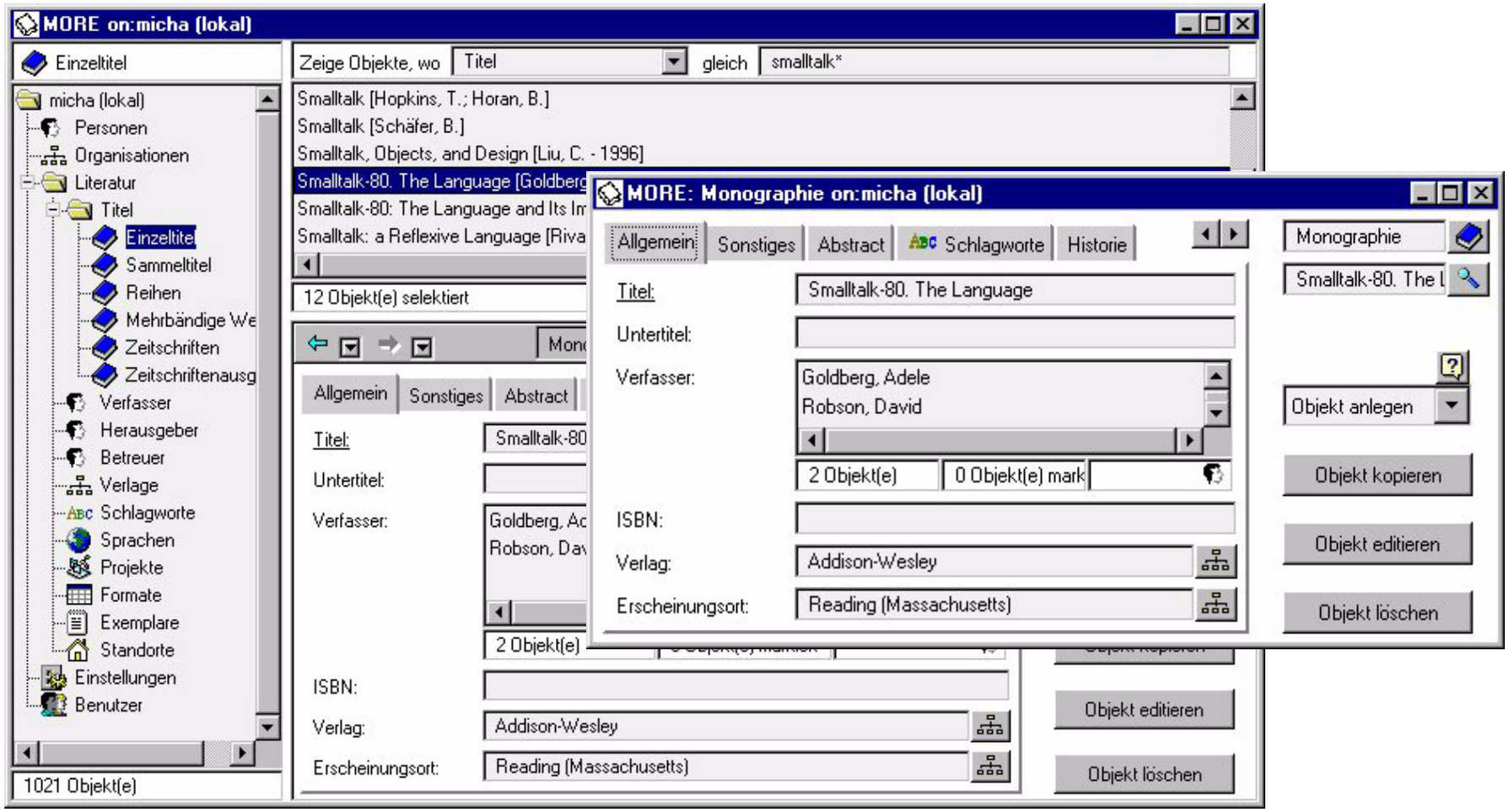
- Zugehörige Methode erzeugt das ValueModel für die Instanz des sub-applicationModel, das für die Koordination des subcanvas zuständig ist.




```
subapp
    "This method was generated by UIDefiner."
^subapp isNil
    ifTrue: [subapp := CommentBrowser2 new]
    ifFalse: [subapp]
```

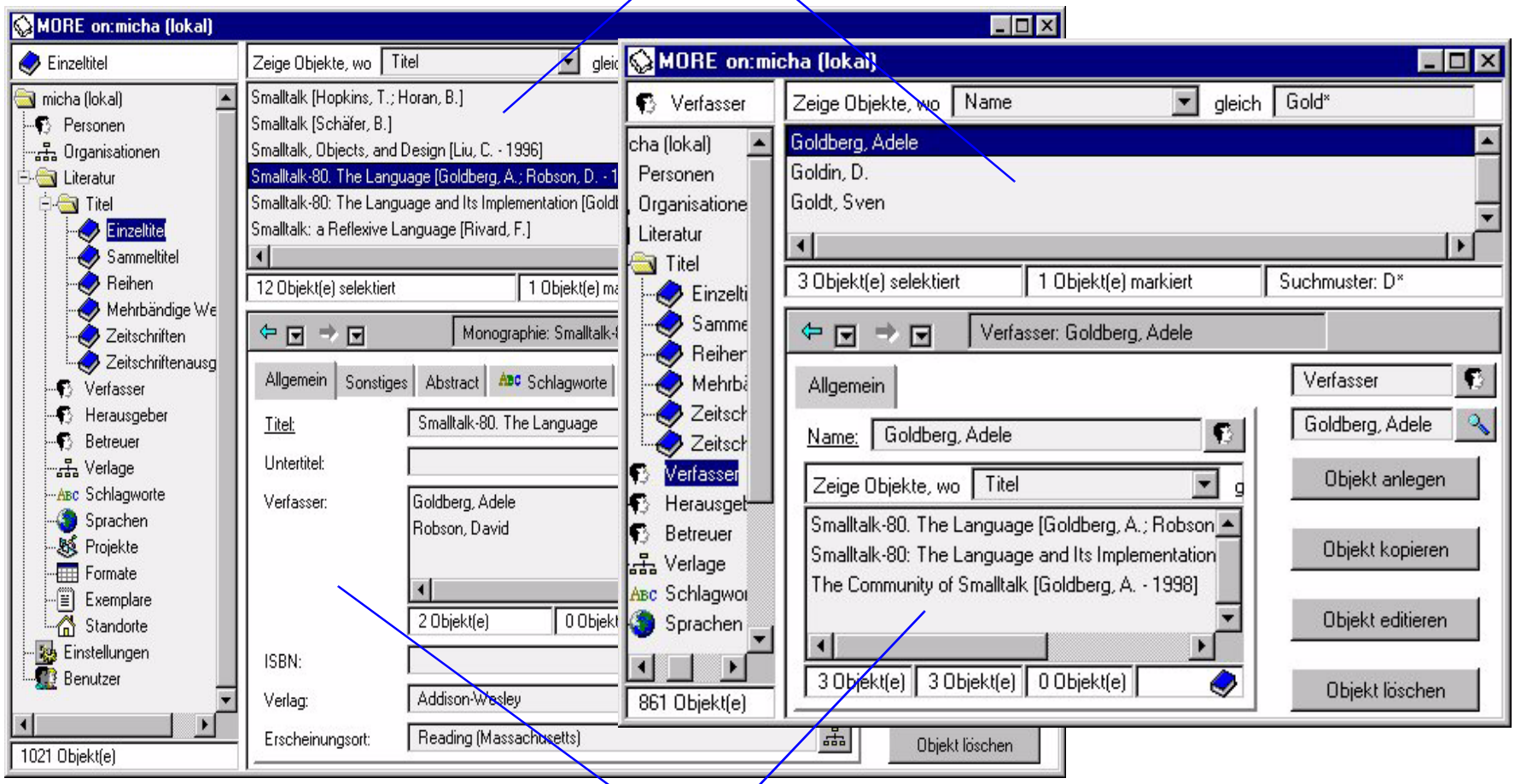
Beispiel: MORE (Universität Koblenz-Landau)

- Einbetten des Literatur-Editors in das Hauptfenster
- ermöglicht die integrierte ***und*** separate Verwendung des Editors



Beispiel: MORE

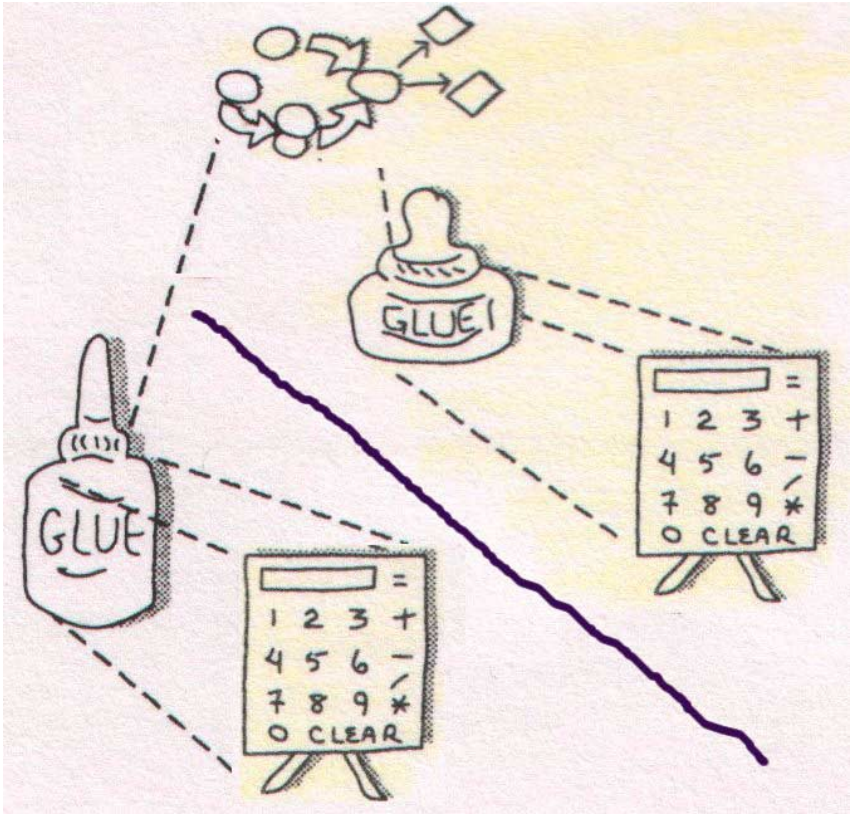
dynamisch änderbare Listen



dynamisch änderbare Editoren

Canvas-Reuse (Neuzeichnen in anderem ApplicationModel)

copy & paste



Reuse:
nur Canvas-Idee

- Zum Ausprobieren noch akzeptabel
- Für langfristige Entwicklungen - ***NEIN*** !!!

(in Anlehnung an
[VW 1992], S. 69 ff.)

☛ Redundanz, Wartungsprobleme

Canvas-Reuse als Subcanvas

- Kein einfaches Kopieren des canvas, sondern Integration als subcanvas

➡ direkt betroffen von zukünftigen Änderungen des subcanvas

- ABER: Kein Nutzen des alten Original-ApplicationModel

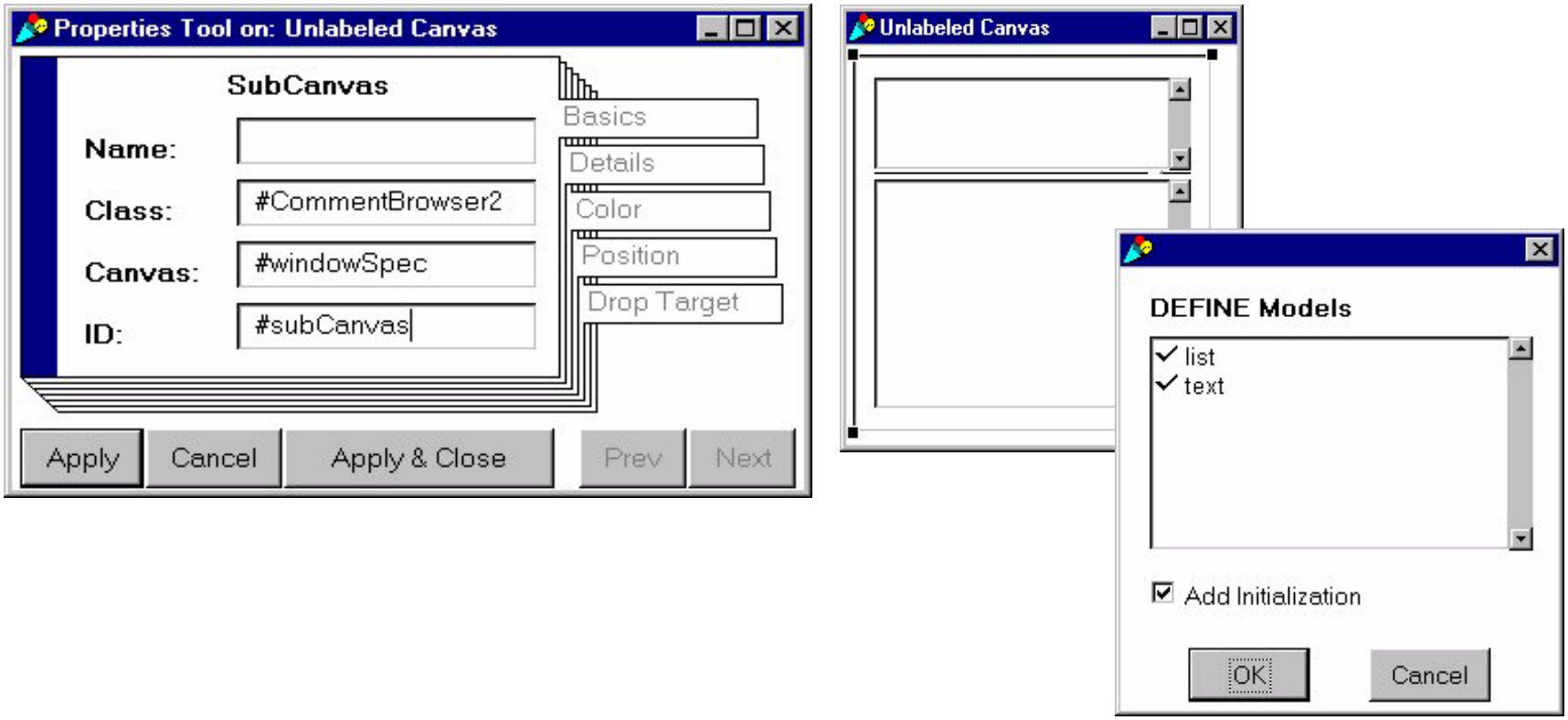
➡ Nachteil:
  - Neuimplementierung des „glue“
  - keine automatische Anpassung des neuen „glue“ bei Änderung des subcanvas

➡ Vorteil:
  - Überschreiben, neue Funktionalität mit demselben Interface

Reuse: Canvas

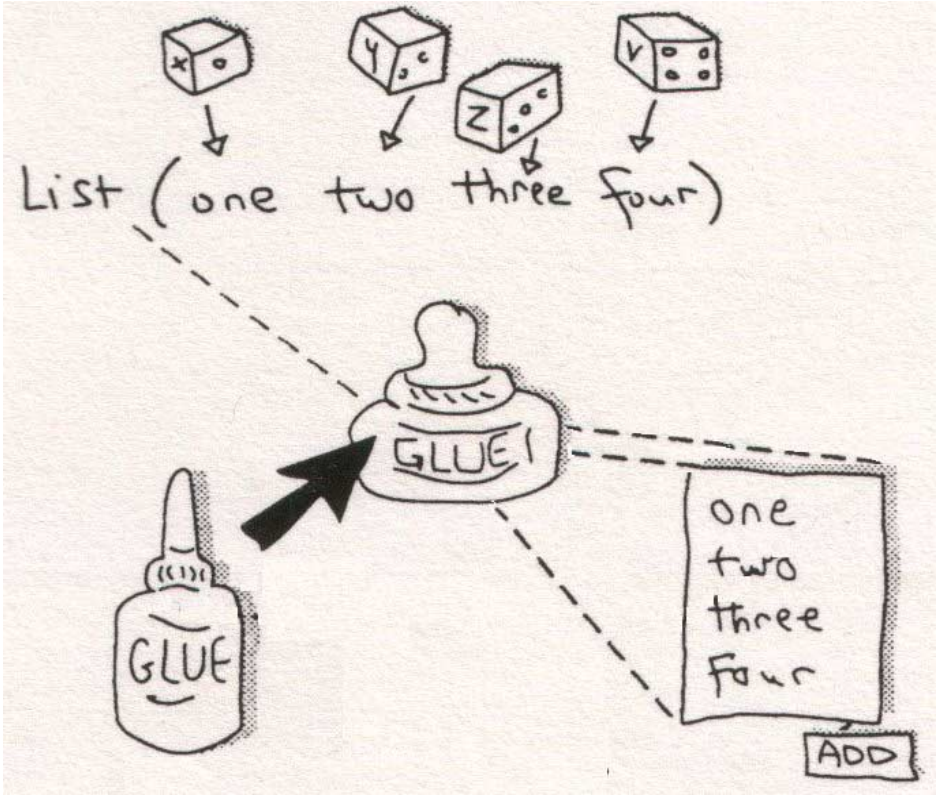
(in Anlehnung an [VW 1992], S. 69 ff.)

Beispiel: Canvas-Reuse des CommentBrowser2



- Client-Aspekt (Name) bleibt leer.
- nur Angabe von Klasse und verwendetem canvas.
- Neues Applikationmodell muß die Aspekte des subcanvas selbst definieren.

Canvas und ApplicationModel Reuse durch Vererbung

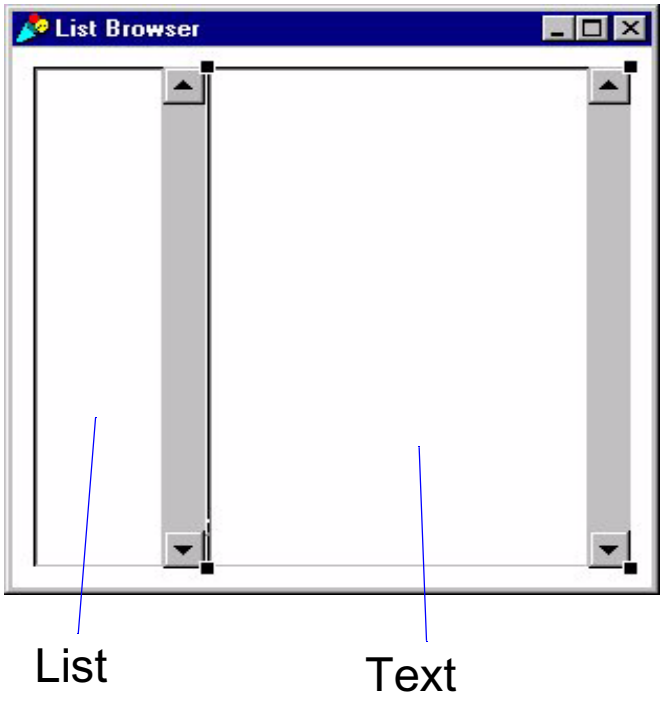


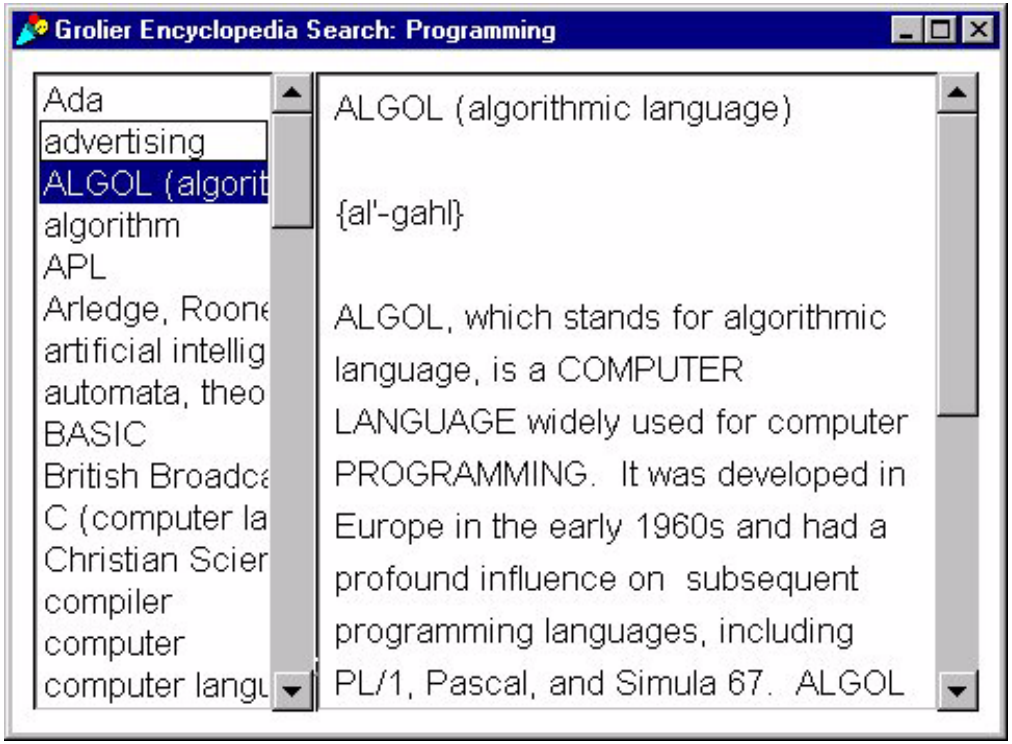
Reuse:
canvas und altes application model

- Anpassung des „glue“
  - z.B. zusätzliche Domänenmodelle
- kein Neuzeichnen des canvas

(in Anlehnung an [VW 1992], S. 69 ff.)

Beispiel: Grolier Encyclopedia (Simple Text Interface)

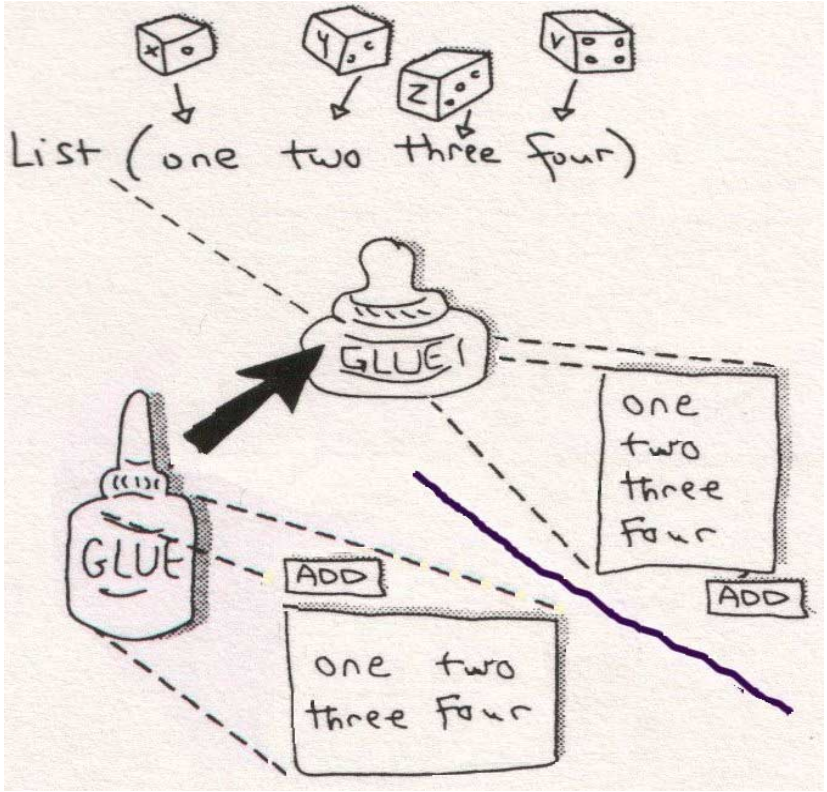




GrolierReader <: ListBrowser2

- übernimmt windowSpecs
- Anpassung der Methoden für Textzugriff: getTextFor:aKey
- Erweiterung um zusätzliches GrolierEncyclopedia-Object

ApplicationModel Reuse durch Vererbung

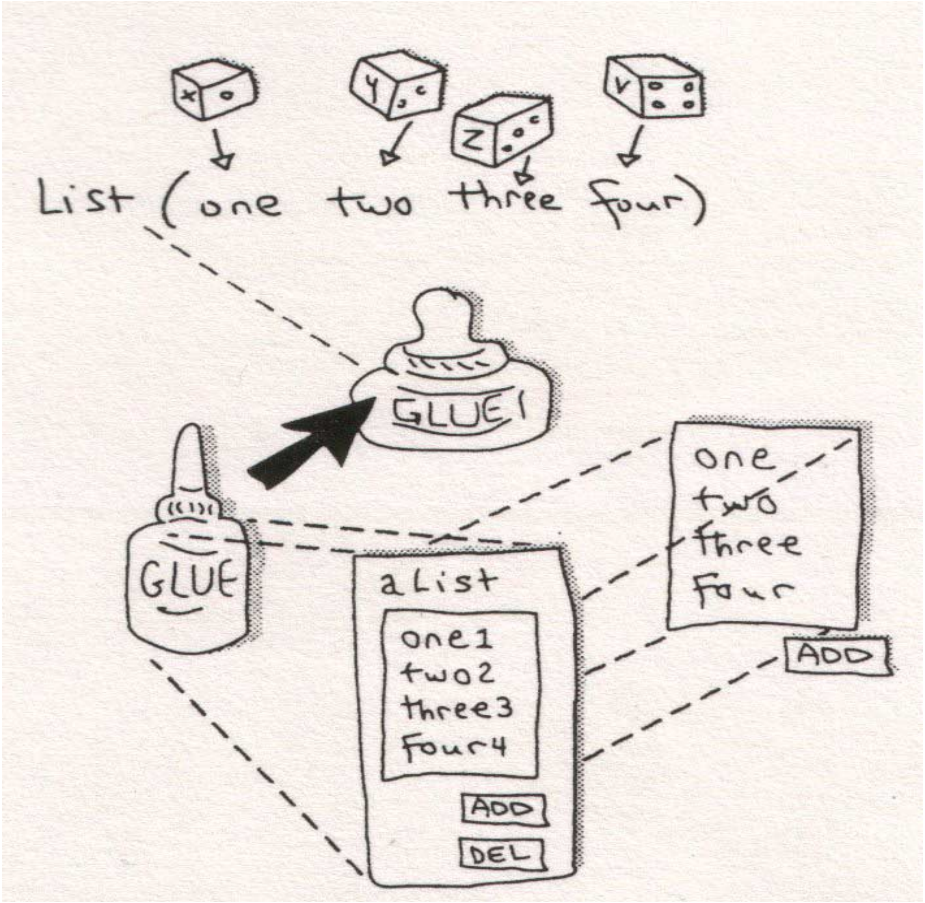


Reuse:
altes application model

(in Anlehnung an [VW 1992], S. 69 ff.)

- Anpassung des „clue“ (z.B. zusätzliche Domänenmodelle)
- Ersetzen des geerbten Canvas (keine Kopplung zwischen neuer und alter GUI)
  ☛ Problematisch für Wartung (vgl. Copy & Paste)
- Anpassung spezieller Interface-Specs in Unterklassen (filterSpec ...)

Canvas und ApplicationModel Reuse durch Subcanvas und Vererbung

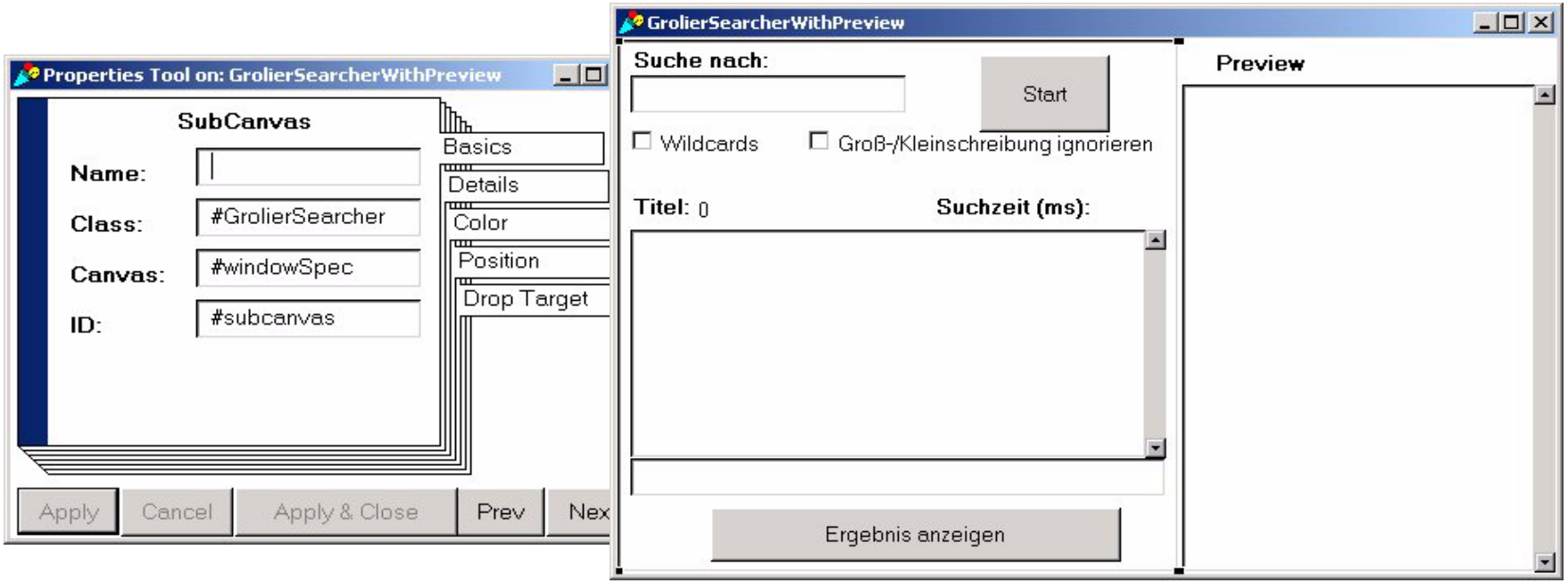


Reuse:
canvas und altes
application model

([VW 1992], S. 74)

- Anpassung des „glue“ (z.B. zusätzliche Domänenmodelle)
- Integration des geerbten canvas als subcanvas

Beispiel: GrolierSearcherWithPreview>>windowSpec

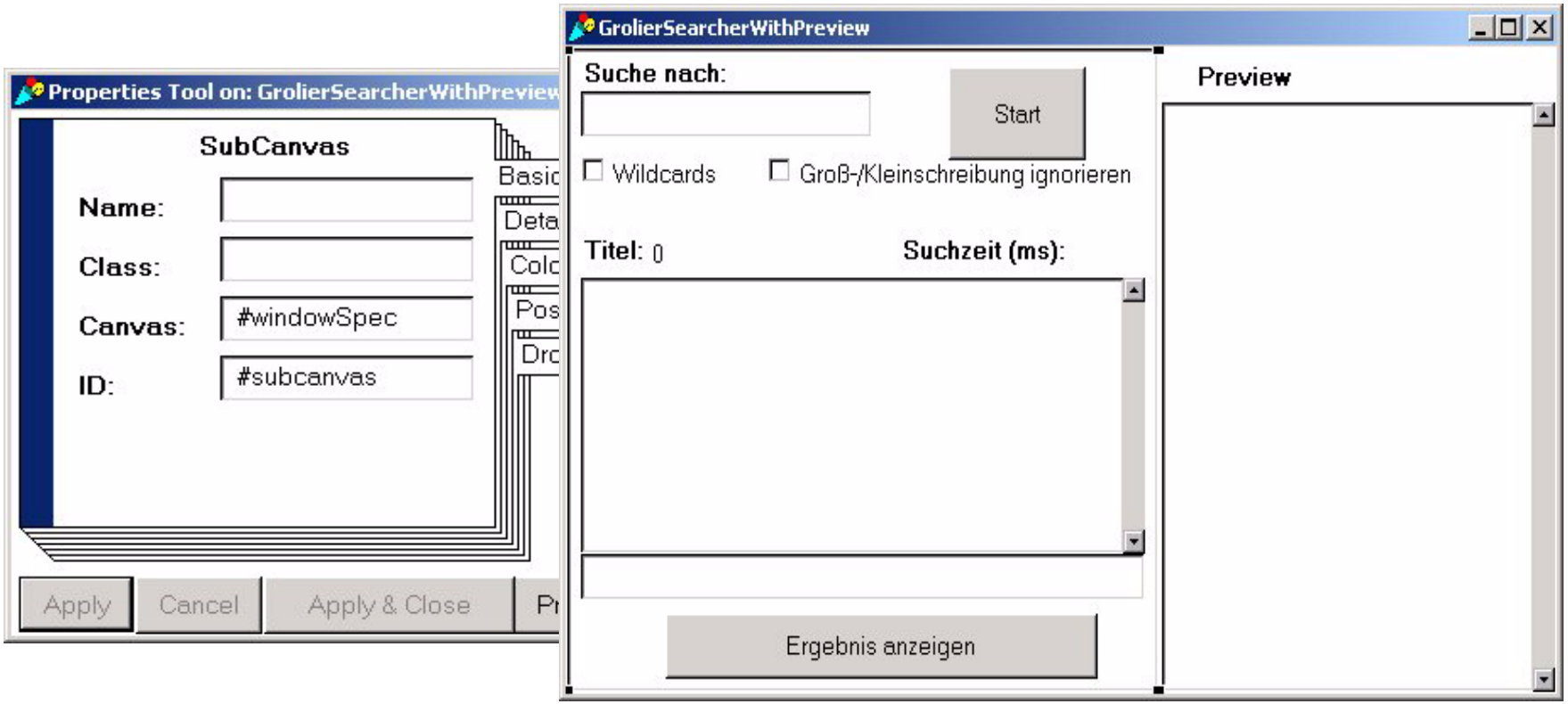


- Angabe des Namens der überschriebenen Interface-Spec
- Explizite Angabe der Oberklasse

- Kopplung zwischen alter und neuer Interface-Spec
- Erweiterung der alten Interface-Spec mittels Subcanvas-Technik anstatt „super“
- Nur einmalige Wiederverwendung möglich, da „glue“ auch nur einmal geerbt wird.

➡ Simulation von „super“ durch Subcanvas-Komposition bei Interface-Specs

Beispiel: GrolierSearcherWithPreview>>windowSpecWithPreview



- nur Angabe des Namens der Interface-Spec
- Dieser muß verschieden vom Namen der neuen Spec sein.
- Auch in diesem Fall nur einmalige Wiederverwendung

➡ Entspricht eher einer Komposition

➡ Auch anwendbar für die Integration einer eigenen Interface-Spec

Zusammenfassung Canvas-Reuse

- alle drei Applikationsbestandteile sind in unterschiedlichen Kombinationen wiederverwendbar:
    - Domain Model
    - Application Model
    - Canvas
- Reuse-Mechanismen:
    - Komposition: subcanvas
    - Vererbung: Erweiterung + Abänderung
    - pure Canvas-Integration (langfristig ersetzen durch Komposition und Vererbung)
    - KEIN Copy & Paste
- Anwendung:
    - verlangt Vorausplanung (Modularisierung der GUI)
    - Entwicklung generischer Oberflächen - ***Abstraktion***
    - Restrukturierung bei nachträglicher Wiederverwendung
        - ➡ Verbesserung der langfristigen Wartbarkeit

8. Smalltalk-Sprachidiome

8.1. Sprachaspekte

Lazy Initialization

Problem: Verwendung nicht initialisierter Variablen

➡ *UndefinedObject doesNotUnderstand*-Fehler beim Verwenden dieser Variablen
➡ Nil-Problematik objektorientierter Programmiersprachen

Lösung:

- Integration von Zugriff und eventueller Initialisierung

```
Person>>surname
^ surname
```

```
Person>>surname
"answer the surname of the receiver"
"use lazy initialisation"

^ surname isNil ifTrue:[String new]
                ifFalse:[surname]
```

- Verwendung spezieller Null-Objekte (Entwurfsmuster) anstatt von nil

Verwendung von *Exception Handling* anstatt expliziter Fehlerobjekte

Verwendung von Fehlerobjeken

- Rückgabe eines Fehlerobjekts anstatt des Ergebnisses

➡ vergleichbare Situation wie bei der Nil-Problematik

➡ ständige Abfragen (isXXXError) auf den Fehler bei allen direkten und indirekten Klienten, die das Ergebnis dieser Methoden verwenden.

Exception Handling von Visualworks

Signaling Exceptions

```
Error raiseSignal
```

Exception Handling

```
[... some work ...]
on: Error
do: [ :ex | Error Handling]
```

Douple-Dispatching

- spezielle Technik im Single-Dispatch Kalkül zur Simulation eines Multiple-Dispatch-Kalküls mit Überladen.

- Alternative zu case-statements zur Typabfrage

Bsp: Arithmetik

```
Integer >> + aNumber
    "Answer the sum of the receiver and the argument, aNumber."

    ^aNumber sumFromInteger: self

FixedPoint >> + aNumber
    "Answer the sum of the receiver and the argument, aNumber."

    ^aNumber sumFromFixedPoint: self
```

Nachteil: höhere Komplexität

Entkopplung von globalen Bezeichnern

- direkte Verwendung von Klassenbezeichnern
  - ➡ Abstraktion durch spezielle Methoden - defaultControllerClass
- direkte Verwendung fest codierter Strings (Konstanten)
  - ➡ Abstraktion durch Symbol-String-Dictionaries
  - ➡ Unterstützung verschiedener Sprachen
  - ➡ Abstraktion durch Methoden
- Berechnung von Methodenbezeichnern und Bezeichnern für graphische GUI-Elemente aus fest codierten Strings.
  - ➡ Abstraktion durch Methoden
- direkte Verwendung von globalen Variablen
  - ➡ Abstraktion durch Singelton-Muster

➡ Anstatt des konkreten Bezeichners sollte ein Alias verwendet werden, der nur an einer Stelle definiert ist.
  - ➡ leichtere Änderbarkeit
  - ➡ kein „Vergessen“ von Aufrufpunkten -> Robustheit

Verwendung von Template-Methoden

```
App>>enableButtons
    self addButton enable: true.
    self deleteButton enable: self itemSelected.

App1>>enableButtons
    self addButton enable: true.
    self deleteButton enable: self itemSelected and:[self selectedItem removable].
```


Änderungen in App müssen in allen Unterklassen manuell nachgezogen werden !!!

Redundanz

```
App>>enableButtons
    self addButton enable: self addButtonCondition.
    self deleteButton enable: self deleteButtonCondition.

App>>addButtonCondition
    ^ true

App>>deleteButtonCondition
    ^ self itemSelected

App1>>deleteButtonCondition
    ^ super deleteButtonCondition and:[self selectedItem removable]
```

Fehlendes Super

```
A >> initialize
    a := 0.

B >> initialize
    b := 0.
```

„Erbe ausschlagen?“

```
A >> initialize
    a := 0.

B >> initialize
    a := 0.
    b := 0.
```

Redundanz

```
A >> initialize
    a := 0.

B >> initialize
    super initialize.
    b := 0.
```

richtig

Fehlendes yourself

- yourself ist eine spezielle Methode, die den Empfänger zurückgibt.
- Am Ende einer Kaskadierung erforderlich, wenn der Empfänger der Kaskade benötigt wird.

```
Set new
    add:1; add:2; add:3;
    yourself
```

8.2. Modellierungsaspekte

Vermeidung von Redundanz

für Langzeitentwicklung: KEIN Copy & Paste

➡ massive Code-Redundanz

➡ mehrere ähnliche Implementierungen

➡ unvollständige Wartung, Erweiterung und Fehlerbehebung

Stattdessen: **Nutzen der Abstraktionsmittel der Objektorientierung**

- Objektkomposition
  - Entwurfsmuster
  - Delegation
- Vererbung

Refactoring

Dokumentation - Kommentare, Modelle und Spezifikationen

- literate programming
- Spezifikation

Hintergrund:
- langlebige Software
- Wiederverwendung
- wechselnde Entwickler

Klassen:
- Zweck, Verwendungskontext (Rollen in Entwurfsmustern)
- Klasseninvarianten, Klassenstruktur (Instanzvariablen, Klassenvariablen)

Methoden:
- Zweck, Vor- und Nachbedingungen, Ausnahmen, Aufrufkontext
- Design By Contract

Refactoring und Extreme Programming

Bei Weiterentwicklung können ehemals abstrahierte Aspekte plötzlich wichtig werden.

➡ Refactoring: Restrukurierung des Systems unter weitgehender Erhaltung der Semantik.

Voraussetzungen:
➡ existierende Spezifikationen für Klassen und Methoden
➡ automatische Tests mit hinreichender Code-Abdeckung

Refactoring und Testen bilden eine Einheit!!!

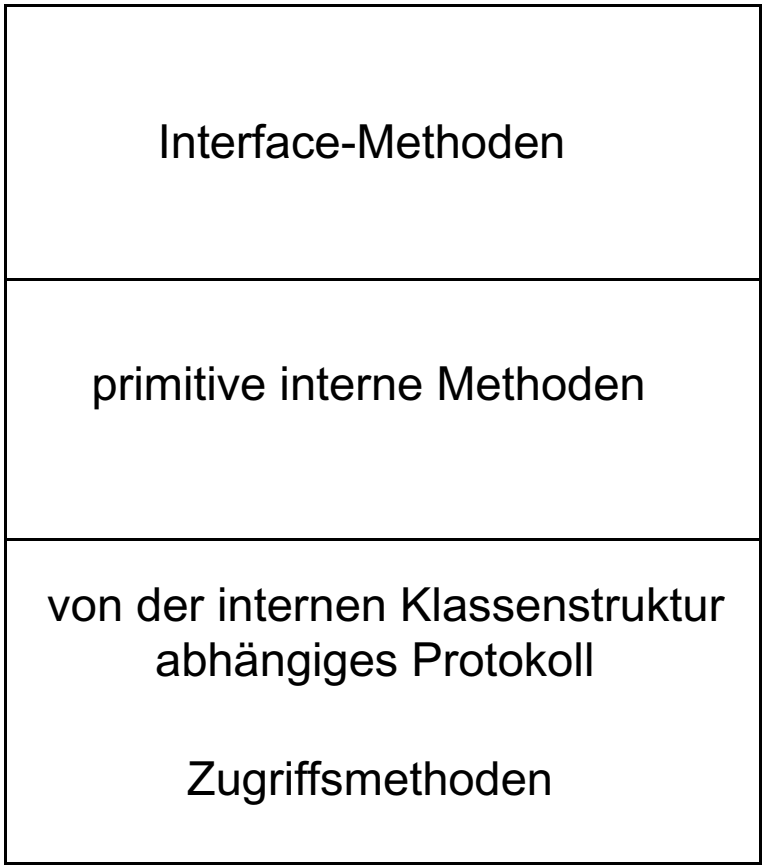
- parallele Entwicklung von Implementierung und Tests
- spezielle Testklassen für die einzelnen Anwendungsklassen
- Testframeworks (SUnit)

- Code-Reviews und Pair-Programming als weitere Qualitätssicherungsmaßnahmen.

Klare Modularisierung

- Zerlegung der Anwendung in separate lose gekoppelte Module
  - Trennung zwischen Fachlogik und GUI
  - Fasadenmuster
  - Fabriken
- Verwendung von Event- und Dependency-Mechanism zur Kopplung
  - implizite Kopplung
  - Beobachter-Muster
- schmale, übersichtliche Klassen
  - Rollenkonzept
  - Zerlegung komplexer Objekte des Anschauungsbereiches in eine Gruppe einfacherer Objekte
- enge wohldefinierte Schnittstellen
- kurze Methoden
  - ein Aspekt/Aufgabe

Schichtenstruktur für Methoden innerhalb einer Klasse



Methoden mit einfachen Vorbedingungen

interne Methoden mit speziellen Vorbedingungen (Verknüpfungsmethoden für bidirektionale Beziehungen)

Einfache Änderungen der Klassenstruktur haben nur Auswirkungen auf diese Schicht.

9. Sprachtheorie für Smalltalk

9.1. Sprachmodell

objektunterstützend

- Unterstützung von Objekten als Menge von Variablen **und** Methoden

klassenunterstützend

- Unterstützung von Klassen als Beschreibung von Objekten
  ➡ kein Methodenaustausch, wie in einigen klassenlosen Kalkülen.

- Speichern der Methoden bei der Klasse
  ➡ Delegationsmodell
  (Alternative: Speichern der Methoden direkt im Objekt
  ➡ Einbettungsmodell)

- keine primitiven Basistypen

nicht statisch typisiert

Vererbung

- Einfachvererbung
- Instanzvariablen dürfen nicht überladen werden.
  ➡ flacher Gültigkeitsbereich
- Methoden dürfen überschrieben, aber nicht überladen werden.
  ➡ verschachtelter Gültigkeitsbereich
- dynamisches (spätes) Binden von Methodenaufrufen
  ➡ Inklusionspolymorphismus
- baumartige Vererbungsstruktur mit der zentralen Oberklasse Object

Single-dispatch Kalkül

- Auswahl einer Methode erfolgt nur auf Basis des Empfängers

  Alternativen:
  Multiple-dispatch Kalkül: Alle Argumente (kein ausgezeichneter Empfänger)
  selective Multiple-dispatch Kalkül: Empfänger + Argumente

Imperatives Sprachmodell

Klassifizierendes Merkmal für imperative Sprachen:

- Interpretation von Variablen als Speicherbehälter
- Existenz einer Zuweisungsoperation zur Änderung dieser Behälter

➡ Smalltalk ist eine imperative Sprache.

- „Programmieren im Kleinen“ (Methoden) vergleichbar zu prozeduralen Sprachen
- zustandsveränderliche Objekte
- Objektidentität zur Unterscheidung zustandsgleicher, aber verschiedener Objekte

Alternative: Funktionales Sprachmodell

- keine zustandsveränderlichen Objekte
- Verzicht auf Instanzvariablen möglich
- Ausdrücken des unveränderlichen Objektzustands durch konstante Methoden
- keine Objektidentität

Referenzsemantik

- mehrere Aliase (Referenzen) auf ein Objekt (vgl. Zeigervariablen)

9.2. Sprachreflexion

- Organisation: Klassen, Methoden und Botschaften in Smalltalk ausgedrückt.
- Compiler und Laufzeitinterpreter in Smalltalk realisiert.

➡ Abfragen und Abändern von Klassen zur Laufzeit mit Smalltalk-Mitteln

- Erzeugen von Klassen
  subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- Variablen
  instVarNames, classVarNames, addInstVarName:, removeInstVarName:
- Methoden
  selectors, addSelector:withMethod:category:
- Botschaften, Klassenzugehörigkeit
  respondsTo:, canUnderstand
- Klassenzugehörigkeit
  isKindOf:, isMemberOf:, class, isMeta

Indirektes Ausführen von Methoden

perform: aSymbol

"Send the receiver the unary message indicated by the argument. The argument is the selector of the message."

perform: aSymbol with: anObject

"Send the receiver the keyword or binary message indicated by the argument. The first argument is the selector, the other argument is the argument of the message."

perform: selector withArguments: anArray

"Send the receiver the message indicated by the arguments. The argument selector is the selector, the arguments of the message are the elements of anArray."

PluggableAdaptor>>getSelector: aSymbol0 putSelector: aSymbol1

```
" Initialize the receiver to act like the old pluggable classes. "
self
    getBlock: [:m | m perform: aSymbol0]
    putBlock: [:m :v | m perform: aSymbol1 with: v]
    updateBlock: [:m :a :p | a == #value or: [a == aSymbol0]]
```

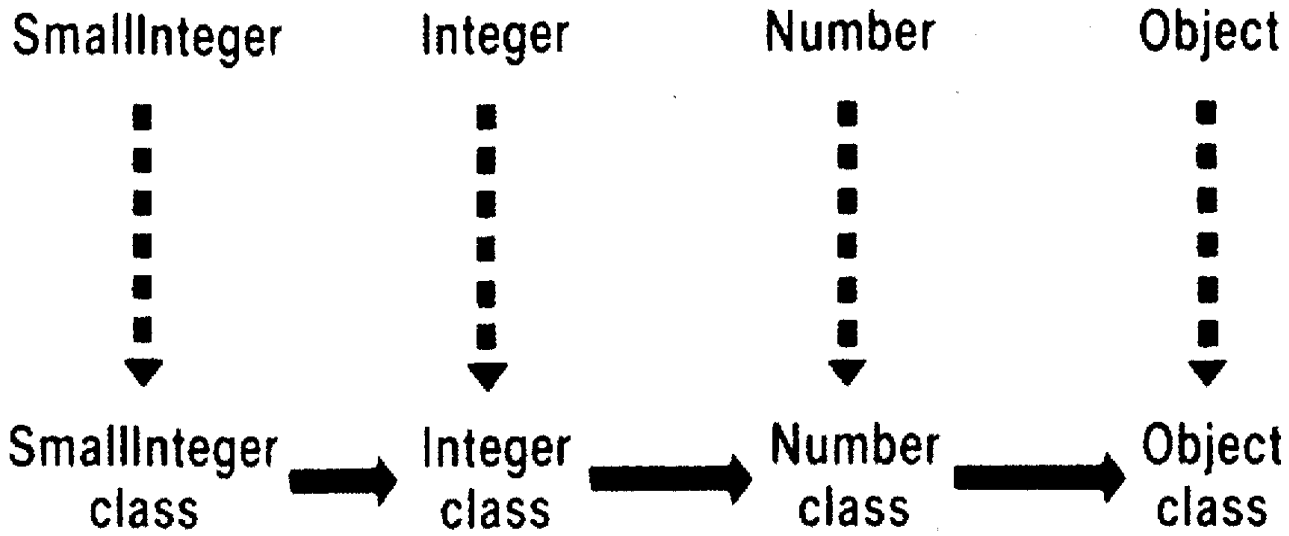
Deklarative Programming

- deklarative Spezifikationen
- Interpretation dieser durch Builder-Klassen und perform-Methoden
- Erzeugen entsprechender Objektstruturen auf Basis der Delarationen

```
ListBrowser2 class>>windowSpec
    ^#(#FullSpec
        #window:
        #(#WindowSpec
            #label: 'List Browser'
            #bounds: #(#Rectangle 337 193 642 477 ) )
        #component:
        #(#SpecCollection
            #collection: #(
                #(#TextEditorSpec
                    #layout: #(#LayoutFrame 10 0 5 0.3 -10 1 -10 1 )
                    #name: #text
                    #model: #text
                                    ... )
                #(#SequenceViewSpec
                    .....) ) ) ) )
```

9.3. Metaklassenstruktur

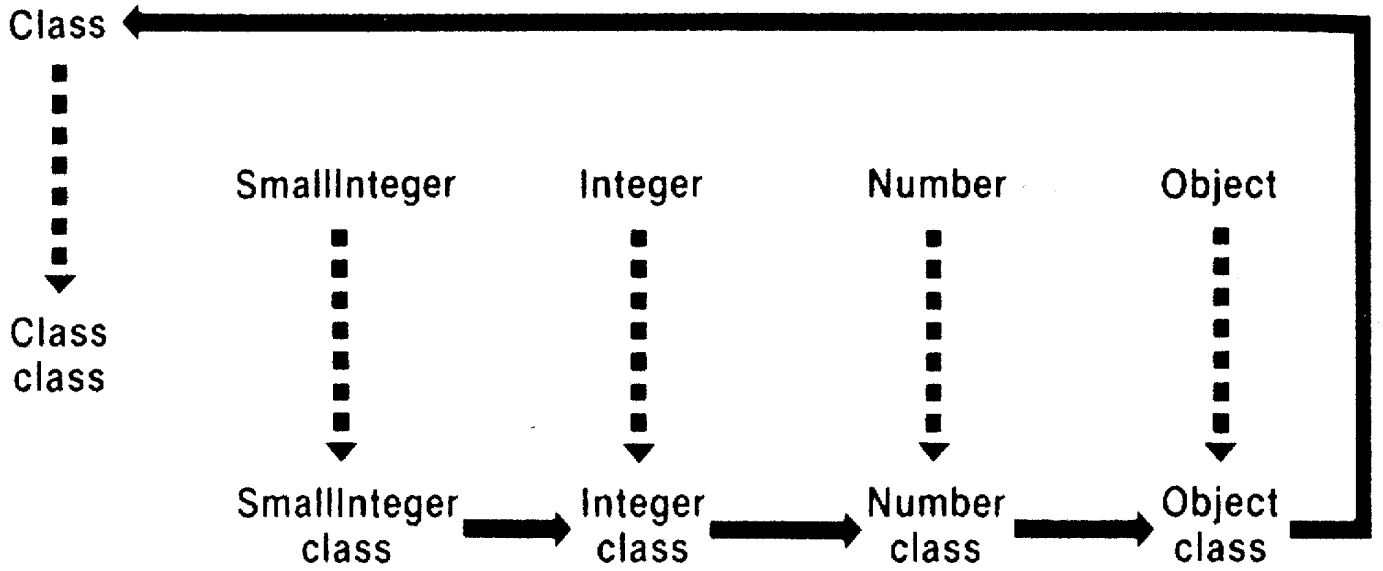
- Jede Klasse hat eine Metaklasse.
- Jede Klasse ist das einzige Exemplar ihrer Metaklasse.



(aus Goldberg, Robson 1989)

- Die Oberklasse von *Object class* ist *Class*. (Beachte *Object* hat keine Oberklasse.)

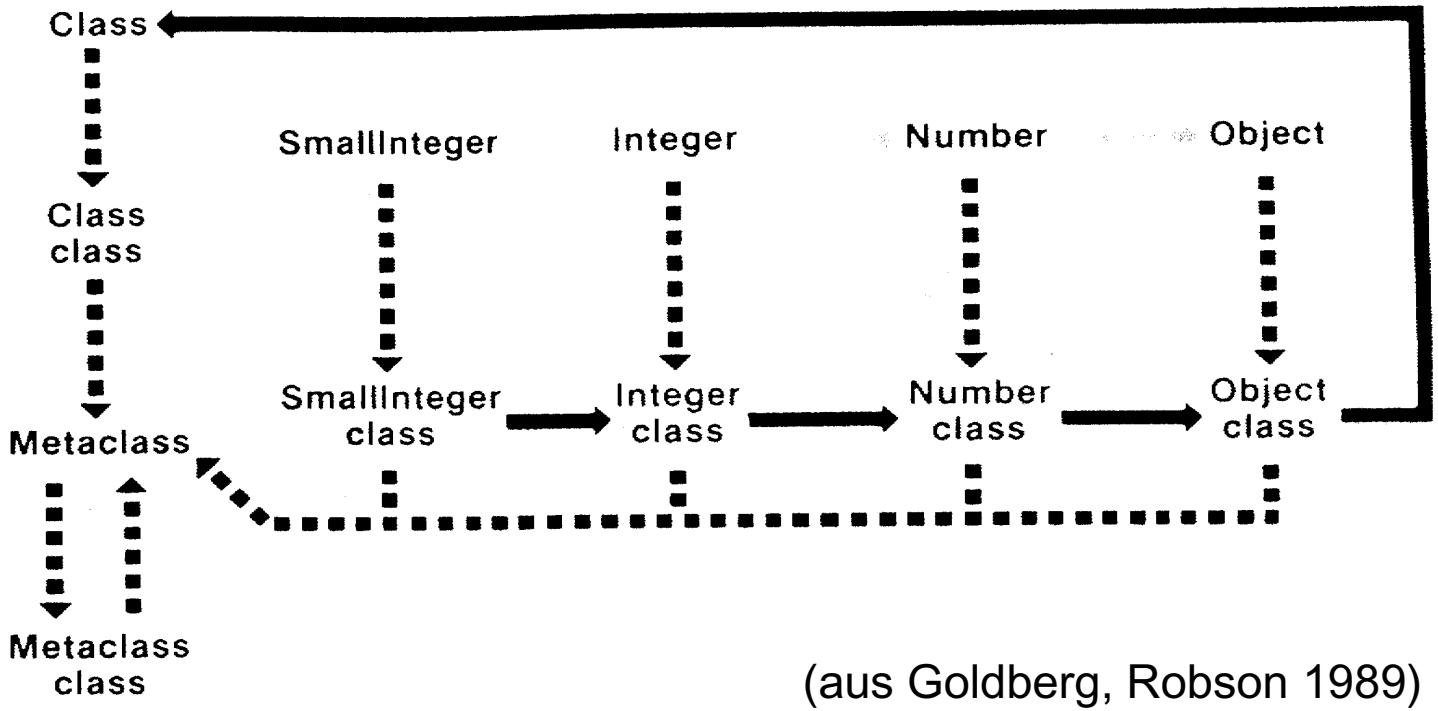
- *Class* ist abstrakte Superklasse aller Metaklassen.
- *Class* selbst ist eine (indirekte) Unterklasse von *Object*.



(aus Goldberg, Robson 1989)

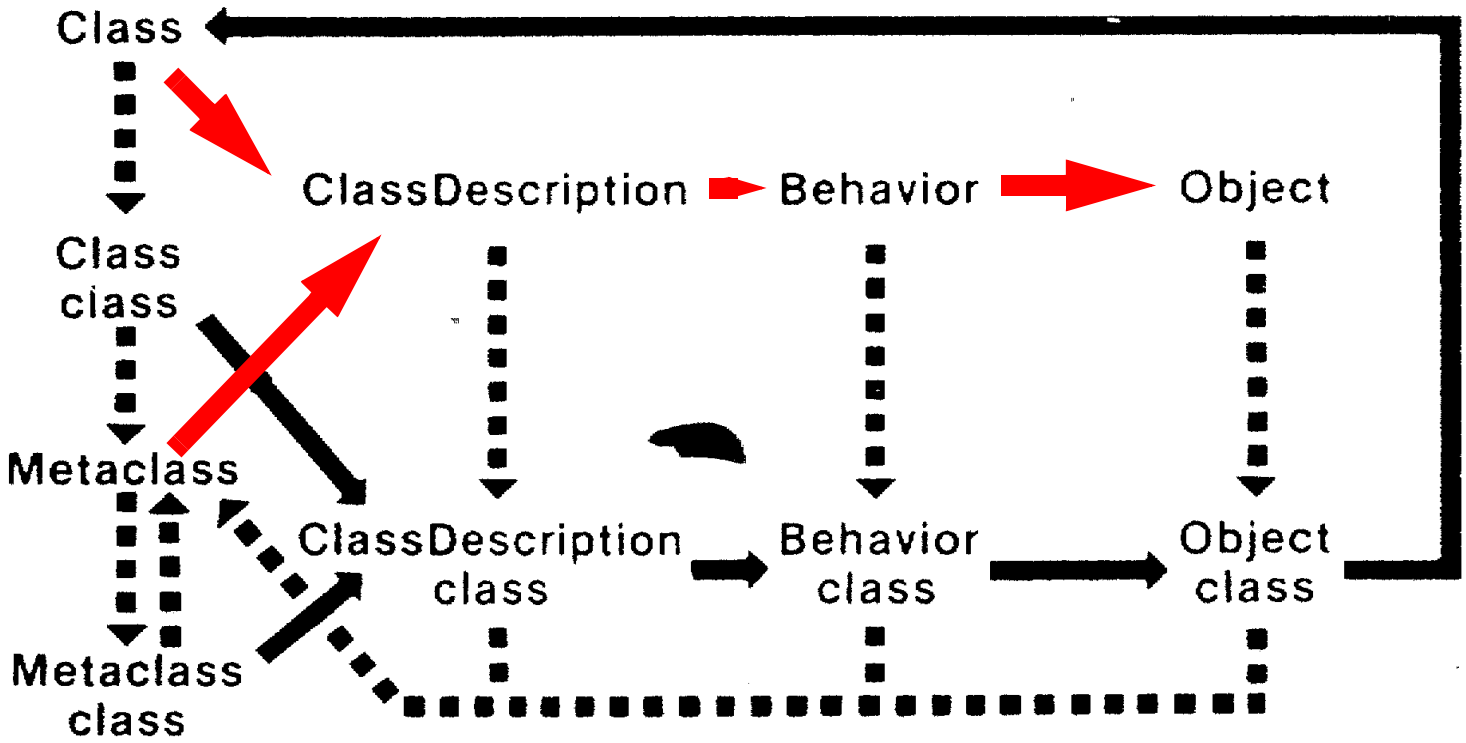
- Jede Metaklasse ist ein Exemplar von *Metaclass.*

- *Metaclass* ist Exemplar von *Metaclass class*.
- *Metaclass* class ist Exemplar von *Metaclass*.



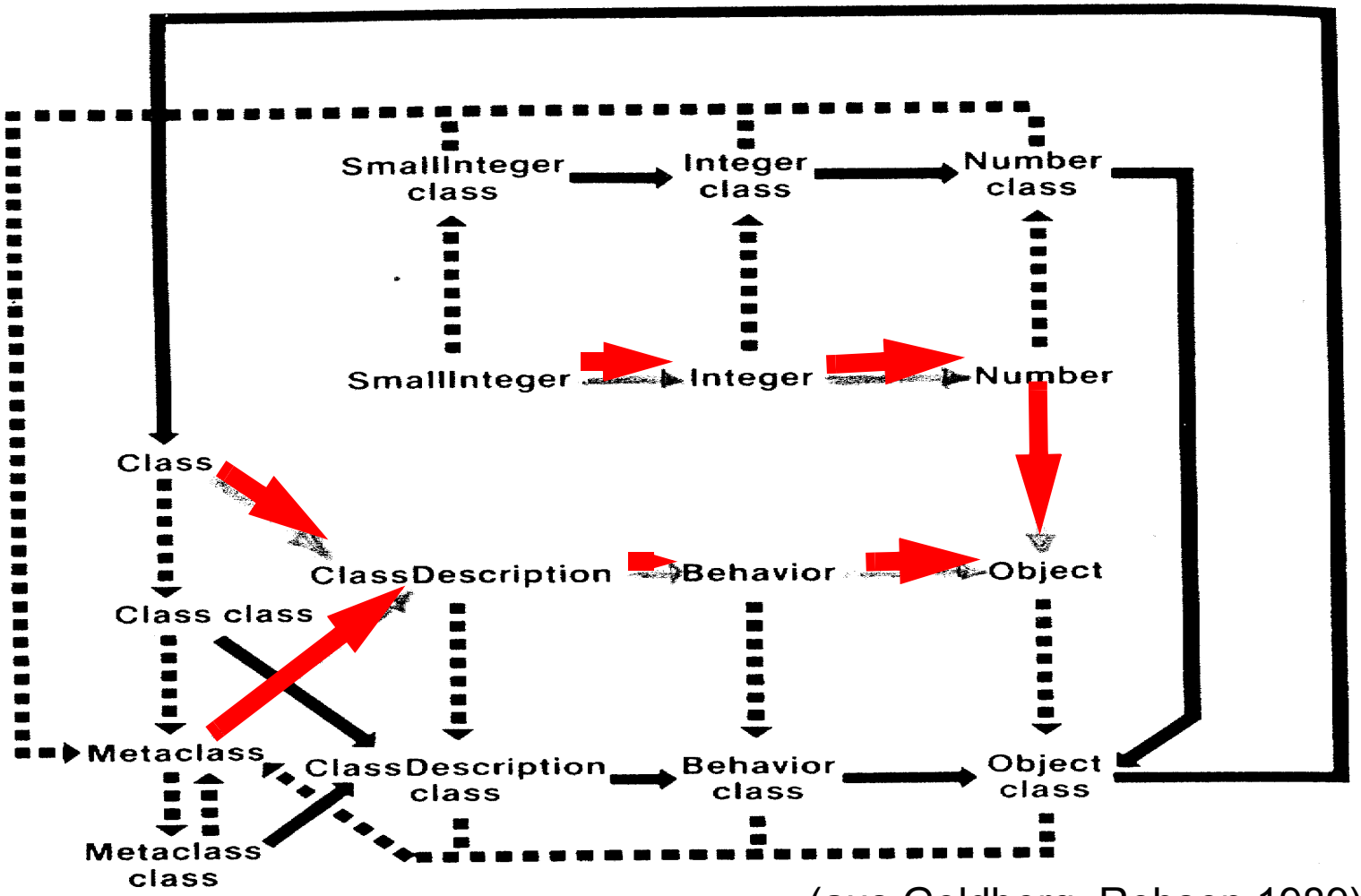
(aus Goldberg, Robson 1989)

- Alle Metaklassen sind in eine Vererbungshierarchie eingebunden.
- Die Vererbungshierarchie der Metaklassen ist parallel zu der Hierarchie der „normalen“ Klassen.



(aus Goldberg, Robson 1989)

- Klassen und Metaklassen der *Number*- und der *Behavior*-Hierachie



(aus Goldberg, Robson 1989)

9.4. Typbetrachtung

Smalltalk ist nicht statisch typisiert.

Probleme:

- doesNotUnderstand-Fehler
- fehlende Typdokumentation
- semantisch fehlerhafte Objektverknüpfungen

Möglichkeiten:

- dynamische Typabfragen
  - Methodenprotokoll
    respondsTo:
    canUnderstand:
  - Klassenzugehörigkeit
    isMemberOf:
    isKindOf:
- Typinferenz (funktionale Programmierung)

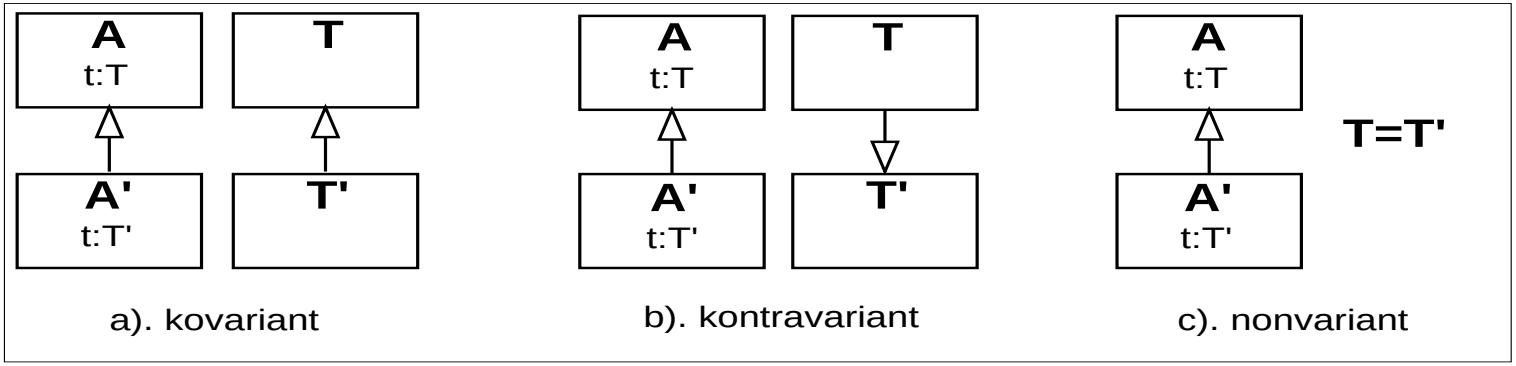
Aber: Objektorientierte Typsysteme gängiger Programmiersprachen sind vielfach nicht ausdrucksmächtig genug.

Beispiel Signaturredeklaration

- Die meisten gängigen objektorientierten Programmiersprachen unterstützen keine Redeklarationen.

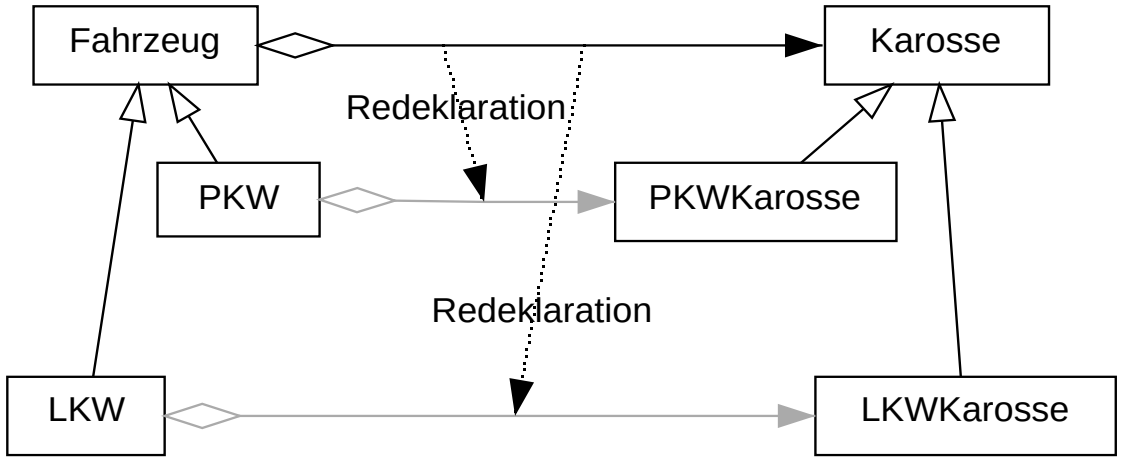
- Aus Sicht der Typtheorie sind folgende Redeklarationen in einem Single-Dispatch-Kalkül möglich:
  - Argument: kontravariant
  - Resultat: kovariant
  - Variable: nonvariant

- Definitionen für Ko-, Kontra- und Nonvarianz
  - gegeben A' <: A mit t:T in A und t:T' in A'



Signaturänderungen aus Modellier- und Programmiersicht

- Kontravariante Argumentänderungen (Bereichserweiterungen) sind praktisch unrelevant.
- Kovariante Einschränkung für Argumente und Variablen ist eine vielfach benötigte Modellierungsmöglichkeit.



Ko- und Kontravarianz-Problematik

- Die gewählte kovariante Redeklaration der Aggregationen kann jedoch zu Typfehlern führen:

[Fahrzeug |
montiere=σs|λ(x:**Karosse**)]

[PKW <c Fahrzeug |
montiere=σs|λ(x:**PKWKarosse**)]

[LKW <c Fahrzeug |
montiere=σs|λ(x:**LKWKarosse**)]

Beispiel:

f:Fahrzeug, k:Karosse
f:=newObjectOf(LKW).
k:=newObjectOf(**Karosse**).

f.montiere(k)

Die Kovarianz-Kontravarianz-Problematik besteht darin, daß im betrachteten Sprachmodell die erlaubten Redeklarationen mit den bei der Anwendung erwünschten Redeklarationen im Konflikt stehen.

Smalltalk und kovariante Redeklarationen

- „angenommene“ kovariante Redeklarationen im MVC-Paradigma.
- „angenommene“ kovariante Redeklaration bei Fabrikmethoden

Warum funktioniert dies?

- disziplinierte Verwendung
  ☛ Kann aufgezeigte Typfehler nicht ausschließen !
- Sicherung der Typinformation durch den Verwendungskontext
- monomorpher Einsatz von Variablen
- Typpropagation, indem notwendige Typentscheidungen dynamisch auf Basis der beteiligten Objekte getroffen werden.

➡ Eine statisch nicht typisierte Sprache kann bei der Entwicklung neuer Typisierungskonzepte helfen.