Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

17.1.2005

Plan 2. Semesterhälfte

- Klassenbasierte dynamische Sprachen
  - ✓ Objective C
  - Smalltalk
- Objekt-basierte Sprachen
- Non-mainstream OO
- Typsysteme für OO

Smalltalk

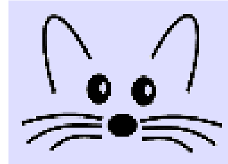
- Smalltalk ist vielleicht *die* OO-Sprache
- Smalltalk-80 bekannteste Variante
  - Entwickelt am Xerox PARC (Palo Alto Research Center)

    In the early 1970's, the Xerox PARC Learning Research Group began work on a vision of the way different perople might effectively and joyfully use computing power.
  - Adele Goldberg, David Robson: "Smalltalk-80: The language", Addison-Wesley, 1989
- Einfluß auf
  - Die späteren OO-Sprachen (Objective-C, Java, C++, Self, JavaScript)
  - OO-Design: Das MVC-Pattern wurde in Smalltalk erfunden (!)

Smalltalk ist anders

- Die Programmierung ist immer interaktiv
  - Lesen & Ändern aller (!) Klassen im System
  - Hinzufügen & Löschen von Klassen & Methoden interaktiv
  - Compilierung in VM Bytecodes automatisch
  - Interpreter für interaktive Eingabe von Ausdrücken
  - Es gibt keine standardisierte Syntax für "Übersetzungseinheiten" (!)
- Das System ist die Dokumentation
  - Jede Klasse und Methode enthält Kommentare am Anfang
  - Der Browser erlaubt interaktiven Zugriff auf alle Klassen
- Der Zustand des Systems wird im (binären) *image* gespeichert
  - Das gesamte System
  - Alle eigenen Klassen + Bibliotheksklassen
  - Der aktuelle Zustand der graphischen Umgebung

Smalltalk Verwendung



- Squeak (`http://www.squeak.org`)
  ist eine freie Implementierung von Smalltalk-80
- Läuft unter UNIX, Mac OS X, MS Windows
- Binaries sind erhältlich, Compilierung kein Problem
  (Aber Compilierung ergibt nur VM, Images müssen von Hand nachinstalliert werden.)
- Start des Systems
  - `inisqueak` erzeugt initiales Images mit System
  - `squeak` startet VM auf vorhandenem Image
- Bedienung gößtenteils über die Maus (Tastenbelegung non-Standard)

Komponenten des Smalltalk-Systems

- World Menu: Mausklick in den Hintergrund
- Open Menu: Zugriff auf Tools
  - Browser: Klassen des Systems untersuchen
  - Workspace: Interaktive Eingabe und Ausführung von Code.
  - Project: Organisationseinheit für Fenster
    (Achtung: Klassen werden zwischen Projekten geteilt!)
- View: Ein Widget im Smalltalk-80 Graphics System
- Morph: Ein Widget im Squeak/Morphic Graphics System
- Halos: Änderungen an Morphs (rechte Maustaste zeigt sie an)

Demo

- inisqueak
- save
- squeak
- world menu
- browser
- Ausgabe von Klassen mit `printOut` und `fileOut`
  (mit `tr "\r" "\n"` Zeilenumbrüche aus `fileOut` anpassen)

Designprinzip von Smalltalk

Everything is an object

. . . Kann man darauf eine ganze Sprache aufbauen?

Objekte, Nachrichten, Protokolle

- Objekte bestehen aus
  - (Privaten) Daten
  - (Öffentlichen) Operationen
- Eine Nachricht (message) fordert Ausführung von Operation
- Das Empfänger- (receiver-) Objekt entscheidet, welche Operation es für die Nachricht ausführt
- Der Empfänger antwortet auf (responds to) die Nachricht
- Die Menge der Nachrichten, auf die ein Objekt antwortet, ist seine Schnittstelle (interface)

Klassen

- Klassen beschreiben Mengen von Objekten, die Instanzen der Klasse
  - Instanzvariablen (instance variables) speichern private Daten
  - Methoden (methods) definieren Operationen der Instanzen
- Das Protokoll einer Klasse zählt die Nachrichten aus der Schnittstelle ihrer Instanzen auf.
- Nachrichten mit verwandter Bedeutung werden zu Kategorien zusammengefasst
- Klassen sind auch Objekte

⇒ Klassen sind auch *nur* Objekte

Eingebaute Typen

- Laut Designprinzip sind Integers, Floats, . . . Objekte
- Ihre Operationen sind die arithmetischen Funktionen
- Diese kann man aber nicht direkt in Smalltalk hinschreiben
- Lösung: Primitive Methoden
  - direkt von der Smalltalk VM ausgeführte Operationen
  - nicht in Smalltalk selbst definiert
  - Ausführung durch Senden von Nachrichten

Es kann losgehen: Syntax von Smalltalk

- (Ja, mehr brauchen wir nicht!)
- Programme bestehen aus Ausdrücken
  - Literale
  - Nachrichten senden (damit auch Kontrollfluss-Steuerung)
  - Rückgaben aus Methoden
- Dazu noch Variablen
  - Temporäre (lokale) Variablen
  - Instanzvariablen
  - Pools
    - Klassenvariablen
    - Globale Variablen (hier: Nur Klassennamen)
    - Gemeinsame Variablen mehrerer Klassen

Literale

- Zahlen (Ganze Zahlen, Lange ganze Zahlen und Floats)
- Einzelne Zeichen $$c$
- Strings `'`*Zeichen*`'`
  Achtung: Escape ist doppeltes Hochkomma: Länge von `'AA''BB'` ist 5
- Symbole: `#`*Bezeichner* (kanonische Repräsentation von Strings)
- Arrays: `#(3 2 1 'hello, world!')`
- Abgrenzung: $\{e_1 . \cdots e_n .\}$ erzeugt bei Auswertung ein Array mit den Ergebnisssen der Ausdrücke (durch Punkte getrennt)

Variablen

- Bezeichner für Speicherstellen
- Zuweisung
  - v`:=`e
  - $v \leftarrow e$ (geschrieben: v _ e)
- Pseudo-Variablen (nicht änderbar)
  - `nil`
  - `true`, `false`
  - `self` für Zugriff auf Empfänger-Objekt
  - `super` (➔ Subklassen)
- Deklaration
  - Klassen- und Instanz-Variablen: Beim Anlegen der Klasse
  - Temporäre Variablen: Am Anfang der Methode in $|v_1 \ldots v_n|$
  - Parameter in Keyword-Messages (➔ Nachrichten)

Rückgaben

- Ausdruck `^`e beendet Ausführung der laufenden Methode und gibt das Ergebnis von e zurück
- Ganz entsprechend zu `return` in Java,C++,. . .

Nachrichten an Objekte schicken

- Nachrichten-Ausdruck (message expression)

  receiver message

- Varianten
  - Unäre Nachrichten (unary messages):

    receiver identifier

  - Binäre Nachrichten (binary messages):

    receiver operator argument

  - Schlüsselwörter (keyword messages) haben Argumente

    $receiver\ kw_1 : arg_1 \ldots kw_n : arg_n$

- *identifier*, *operator* und $kw_1 : \cdots kw_n :$ heißen Selektor der Nachricht

Objekte erzeugen

- Klassen sind Objekte
  - Sie antworten auf Nachrichten mit Klassenmethoden
  - Erzeugung von Instanzen durch Nachrichten an Klasse

```
a := Array new: 5.
a at: 1 put: 'hello'.
a at: 2 put: 'world'.
a  #('hello' 'world' nil nil nil)

d := Dictionary new.
d at: 'Smalltalk' put: 'cool'.
d at: 'C++' put: 'ugly'.
d  a Dictionary('C++'->'ugly' 'Smalltalk'->'cool' )
```


Sequenz und Cascading

- Sequenzen: Ausdrücke durch "." abschließen / trennen
- Häufig: Mehrere Nachrichten an gleichen Receiver senden
- Begriff: cascading
- Syntax: $r\ m_1$; m_2; $\cdots$
- Äquivalent zu: $r\ m_1.\ r\ m_2.\ \cdots$
- Beispiel: Einträge in ein Dictionary

Demo

- Workspace für interaktive Eingabe
- Variablen: Einfach zuweisen
- Integer-Konstanten
- Integer binäre Ausdrücke
  - 5 + 2 ok
  - 5 + 3 * 2 ➔ 16 statt 12 (!)
- Integer sind Objekte
  - 25 "Inspect it"
  - Browser class im Inspector-Menü

Kontroll-Strukturen

- Betrachte übliche Anweisung `if` a `then` b `else` c
- Code b und c darf nicht sofort ausgeführt werden
- Konventionelle Antwort: Compiler erzeugt Sprünge, die jeweils b oder c auslassen
- Smalltalk-Antwort: b und c sind Code-Blöcke `[` e_1`.` $\cdots$ e_n`. ]`
  - Blöcke enthalten Sequenzen von Ausdrücken
  - Diese werden zunächst nicht ausgeführt
  - Blöcke sind Objekte
  - Die Ausführung wird durch Nachricht `value` an Block angestoßen
  - Ergebnis der Ausführung ist Ergebnis des letzen Ausdrucks

⇒ Vgl. Lazy evaluation / call-by-name

⇒ Smalltalk braucht keine syntaktische Kategorie "Anweisung"

Kontroll-Strukturen

- Smalltalk braucht keine syntaktische Kategorie "Anweisung"
- Idee: Boolean-Werte verstehen Nachricht für Fallunterscheidung

```
(x = 0) ifTrue: [ self error: 'x must not be zero'. ]
        ifFalse: [ ^ 1 / x ].
```

⇒ Der Boolesche Wert wählt den richtigen Block aus

- Vgl. auch Church-Codierung von Booleans als λ-Ausdrücke

$$\texttt{true} = \lambda xy.x \qquad \texttt{false} = \lambda xy.y \qquad \texttt{if} = \lambda bxy.bxy$$

- Hier wird b in der Fallunterscheidung aktiv
- (Beachte: Funktioniert nur für call-by-name / lazy evaluation)

Erinnerung: Closures in OCaml

```
# let f = fun x -> fun y -> x * y;;
val f : int -> int -> int = <fun>
# let g1 = f 4;;
val g1 : int -> int = <fun> (* enthält Bindung x=4 *)
# let g2 = f 5;;
val g2 : int -> int = <fun> (* enthält Bindung x=5 *)
# g1 2;;
- : int = 8
# g2 2;;
- : int = 10
```

- Funktionen sind Werte
- `fun` erzeugt einen Funktionswert = Closure
- Closures enthalten die statische Variablenumgebung, in der erzeugt wurden, sie sind "geschlossen", weil sie alle zur Berechnung notwendigen Variablenwerte selbst enthalten.

Blöcke als Closures

- Blöcke können Parameter haben

$$\texttt{[:}x_1 \cdots \texttt{:}x_n \texttt{ | } e_1\texttt{.} \cdots e_m\texttt{.]}$$

- Aufruf mit passend vielen Argumenten; Nachricht:

$$\texttt{value: } a_1 \cdots \texttt{value: } a_n$$

- Der Code im Block kann auf die Umgebung des Blocks (context) zugreifen

```
sumArray: a ini: ini
 | i |
  i:=ini.
  a do: [ :j | i := j + i ].
 ^ i.
```

- Vgl. Closures in funktionalen Sprachen
- Der Kontext ist wieder ein Objekt

`map` & Freunde in Smalltalk

- Funktionale Sprachen: Funktionen höherer Ordnung auf Listen
- Umsetzung in Smalltalk: Blöcke und Collections
- Klasse `Collection` bietet `collect:` Nachricht; Beispiel Array:

```
#(1 3 4) collect: [ :i | i * 5]. ==> #(5 15 20)
```

- Iteration mit `do:`

```
j := 0.
#(3 7 189) do: [ :i | j := j + i ].
j ==> 199
```

- Auswahl mit `select:`

```
#(18 3 27 25 4 36) select: [ :i | i even].
==> #(18 4 36)
```

Blöcke sind effizient

```
 9 pushTemp: 0
10 pushTemp: 1
11 send: &
12 jumpFalse: 15
13 pushConstant: true
14 popIntoTemp: 2
15 pushTemp: 2
16 returnTop
```

```
test: b and: c
  | e |
  (b & c) ifTrue: [ e := true ].
  ^ e.
```

- Smalltalk Bytecodes für virtuelle (Stapel-) Maschine
- `Temp` sind die lokalen temporären Variablen
- `ifTrue:` wird in normalen `jump` übersetzt
- Es wird kein Block-Objekt erzeugt
- (Hinweis: Nachricht `and:` ist short-circuit Semantik)

Blöcke als Closures sind effizient

```
9  pushTemp: 1
10 popIntoTemp: 2
11 pushTemp: 0
12 pushThisContext:
13 pushConstant: 1
14 send: blockCopy:
15 jumpTo: 24
17 popIntoTemp: 3
18 pushTemp: 3
19 pushTemp: 2
20 send: +
21 storeIntoTemp: 2
23 blockReturn
24 send: callMe:
25 pop
26 pushTemp: 2
27 returnTop
```

```
 sendBlock: recv for: i0
| i |
 i:=i0.
 recv callMe: [ :j | i := j + i ].
^ i.
```

- Block-Code im Methodencode
- Block enthält nur PC für eigene Bytecodes
- Primitivum `blockCopy:` n in `ContextPart` erzeugt Block mit n Argumenten
- Block enthält aktuellen `Context` (ein Objekt!)

Subklassen

- Die Instanzen einer Subklasse sind gleich den Instanzen ihrer Superklasse, außer an speziell angegebenen Stellen
- Subklassen können
  - Instanzvariablen hinzufügen
  - Methoden hinzufügen (= Nachrichten zum Protokoll hinzufügen)
  - Methoden überschreiben (override)
- Jede Klasse ist eine Subklasse von `Object`

Klassen erzeugen

- Klasse `Class` antwortet auf Nachricht

```
subclass:
instanceVariableNames:
classVariableNames:
poolDictionaries:
category:
```

⇒ Eine neue Klasse wird durch Nachricht `subclass:` ··· an ihre Superklasse erzeugt

⇒ Datenstrukturen der Klassenhierarchie existieren zur Laufzeit

⇒ Smalltalk ist sehr dynamisch

Methode für eine Nachricht

- Ein Objekt antwortet auf eine Nachricht, indem es eine Methode ausführt
- Die Methode befindet sich in der Klasse K des Objektes, oder in einer Superklasse
- Nachricht = Selektor + Argumente
- Die ausgeführte Methode wird zur Laufzeit anhand des Selektors gesucht
  - Beginne in der Klasse K selbst
  - Solange die Methode nicht gefunden ist, geht zur Superklasse über
  - Wenn `Object` den Selektor nicht kennt, schicke `doesNotUnderstand:` an das Empfängerobjekt
- `doesNotUnderstand:` in `Object` unterbricht die Ausführung

Nachrichten an `super`

- Nachrichten an `super` haben Empfänger `self`
- Die Suche nach der implementierenden Methode beginnt in der Superklasse der Klasse von `self`
- Sinn: Überschriebene Methoden verfügbar machen

Demo

- Browser für Klassen
- Dictionary suchen in Collection-Kategorie
- Instance Creation der Klassennachrichten
- Neue Klassen anlegen
- Klassen in Dateien exportieren (Mausmenü: `fileOut`, `Changes/fileOut`)

Abstrakte Klassen

- Abstrakte Klassen implementieren einige Methoden nicht vollständig
- Abstrakte Methoden senden per Konvention `subclassResponsibility` an `self`
- `subclassResponsibility` ist in `Object` definiert und führt dort zu einem Fehler

Methodenaufruf mit Selektoren

- s ist Selektor (Symbol): Man kann s an r schicken (definiert in `Object`)
  - r `perform:`s ohne Argument
  - r `perform:`s `with:` a_1
  - r `perform:`s `with:` a_1 `with:` a_2
  - r `perform:`s `with:` a_1 `with:` a_2 `with:` a_3
  - r `perform:`s `withArguments:` a (mit Array a)
- Beispiel:

```
a := Array new: 4.

a at: 1 put: 'hello'. 'hello'

a at: 1.                    ==> 'hello'
a perform: #at: with: 1.    ==> 'hello'
```


Das Protokoll von `Object`

Die Klasse `Object` definiert ein umfangreiches Protokoll, das in allen Objekten zur Verfügung steht: (Auswahl!)

- Klassen-Zugehörigkeit: `class`, `isMemberOf:`, `responsTo:`, `isKindOf:`
- Vergleich: `hash`, `=`, `~=`
- Kopie: `copy`, `clone` (primitive), `deepCopy`, `shallowCopy`, `veryDeepCopy`
- Dependents (Observer Pattern!): `addDependent:`, `removeDependent:`, `dependents`
- Updating: `changed`, `changed:`, (Änderung verkünden), `update:` (auf Änderung reagieren)

Das Protokoll von `Object`

(Fortsetzung)

- Error handling: `assert:`, `error:`, `subclassResponsibility`
- Streaming (Serialisierung): `byteEncode:`, `printOnStream:`
- Message handling: `perform:`, `performWith:`
- Primitiva: `instVarAt:`, `instVarNamed:`
- Testing: `isInteger`, `isFloat`, `notNil`
- System-Support, GUI-Support, Scripting (Squeak-Mechanismus für Nachrichten), . . .

Class, ClassDescription, Behaviour

`Class` Subklasse von `ClassDescription`, Subklasse von `Behaviour`

- Behaviour: Verwaltung von allen Objekten, die Methoden besitzen
  - Datenstruktur für den Interpreter selsbt
  - Methoden-Dictionary
  - Superklasse
  - Methoden für Zugriff auf Vererbungshierarchie (aber `subclasses` nicht implementiert ➔ abstrakte Klasse)
- ClassDescription: Verwaltung von Klassen, Kommentare, Kategorien
- Class: Superklasse aller Klassen im System
  - Verwaltung von Klassen- & Instanzvariablen
  - Vererbungshierarchie

Designprinzip von Smalltalk

Everything is an object

. . . Aber gibt es da keinen Haken?

Klassen als Objekte – Metaklassen

- Klassen sind Objekte
- Alle Objekte sind Instanzen von Klassen

⇒ Von welcher Klasse sind Klassen Instanzen?

- Antwort: Klassen sind Instanzen von Metaklassen
- Definition: "Klassen, deren Instanzen wieder Klassen sind, heißen Metaklassen."
- Metaklassen beschreiben das Protokoll von Klassen
  - Allgemeine Klassenmethoden (nicht üblich in Smalltalk)
  - Erzeugung von Instanzen der Klasse (Klassenmethoden!)
  - Initialisierung der Klassenvariablen (Konvention: `initialize`)
- Jede Klasse hat ihre eigene Metaklasse
- Eine Metaklasse besitzt genau eine Instanz

Metaklassen sind Klassen

- Problem: Was ist die Klasse einer Metaklasse?
- "Metaklassen unterscheiden sich darin von anderen Klassen, daß sie selbst nicht wieder Instanzen einer Metaklasse sind. Stattdessen sind sie Instanzen der Klasse `MetaClass`." (Goldberg/Robson, S.77)

⇒ Zirkuläre Abhängigkeit

- Metaklassen haben keinen Namen
- Zugriff auf Metaklassen über Nachricht `class` an Instanzen, also normale Klassen

Subklassen für Metaklassen

- Die Subklassenhierarchie von Metaklassen verläuft parallel zur Hierarchie ihrer Instanzen
- Metaklassen können wie andere Klassen Methoden hinzufügen und auch überschreiben
- Damit können in Smalltalk Klassenmethoden überschrieben werden (!)

Überschreiben von Klassenmethoden

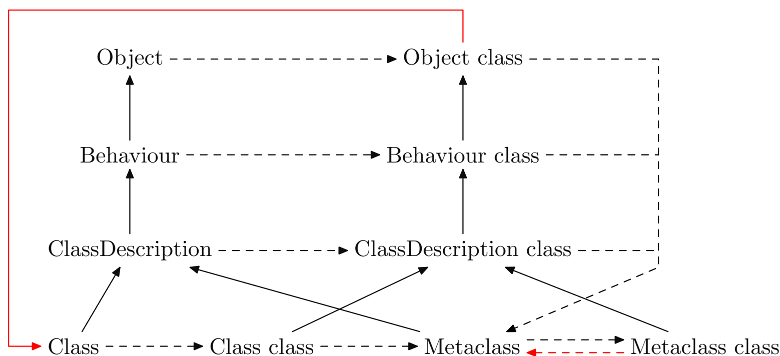
```
Object subclass: #SelfRef
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo'!
!SelfRef class methodsFor: 'test' !
recv: i
 ^ i+ 42. ! !
sendMe
  ^ self recv: 52! !

SelfRef subclass: #SelfRefHeir
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo'!
!SelfRefHeir class methodsFor: 'test'!
recv: i
  ^0.! !
```


Objekte, Klassen, Metaklassen

Goldberg / Robson (Kapitel 16) fassen das Smalltalk-Design in sieben Prinzipien zusammen, aus denen sich logisch die Struktur der Basisklassen ergibt:

1. Jedes Klasse ist letztlich eine Subklasse von `Object`
   Insbesondere ist `Class` eine Subklasse von `Object`
2. Jedes Objekt ist eine Instanz einer Klasse
3. Jedes Klasse ist eine Instanz einer Metaklasse
4. Jede Metaklasse ist eine Instanz von `Metaclass`
5. Die Methoden von `Class` und seinen Superklassen definieren das Verhalten aller Objekte, die auch Klassen sind.
6. Die Methoden von Instanzen von Metaklassen definieren das spezielles Verhalten spezieller Klassen

Der zirkuläre Abschluss



Ausschnitt aus: Goldberg / Robson, Abbildung 16.5

Smalltalk

- Design-Prinzip: Everthing is an object
  - Objekte
  - Klassen
  - Metaklassen
- Objekte beantworten Nachrichten durch Ausführung von Operationen
- Jedes Objekt ist Instanz einer Klasse

⇒ Klassen enthalten Methoden

- Design-Prinzip nicht unproblematisch: Einführung von Metaklassen und Zyklische Abhängigkeiten zwischen Klassen und Objekten des Smalltalk Grundsystems.