

00, ein Streifzug

MARTIN STEFFEN

Januar 1999

Inhalt

- Klassen und Objekte
- Typen, Klassen und Module
- Polymoprhie
- Statik und Dynamik
- Sonstige Features

Inhalt: Der nichtformale Vortrag soll einen Streifzug durch verschiedene Aspekte objektorientierter Sprachen liefern. Nachdem es alles andere als Einigkeit darüber herrscht, was der Kanon der Eigenschaften einer objektorientierten Sprache ausmachen und was nicht, stelle ich eine Auswahl verschiedener Merkmale, die man in objektorientierten Sprachen in Verbindung bringt, vor.

Im ersten Teil **Klassen, Objekte** und **Vererbung** mit **später Methodenbindung** als mehr oder minder unkontroverser Bestandteil der meisten (wenn auch nicht aller) objektorientierter Sprachen. Im zweiten Teil noch unterschiedliche komplexere oder seltenere Sprachmerkmale.

Literatur: Im wesentlichen habe ich folgende Literatur zu Rate gezogen: Budd [Budd, 1997] bietet einen sauberen, nicht zu theoretischen Einstieg in Merkmale objektorientierter Sprachen. Das Standardwerk [Abadi and Cardelli, 1996] zu den Grundlagen von OO bietet eine breit angelegte theoretische Analyse verschiedener objektorientierter Konzepte; die einführenden Abschnitte sind gut lesbar und allgemeinverständlich. Auf vergleichbarem Niveau ist Castagnas Buch [Castagna, 1997]. Es ist insofern ungewöhnlich, als es die theoretische Behandlung von Sprachen mit *multiple-dispatch* behandelt, die nicht zur Hauptrichtung objektorientierter Sprachen gehören.

Für Spezialfragen habe ich noch auf [Cook, 1991], [Bruce et al., 1996a], [Bruce et al., 1996b] zurückgegriffen.

Was ist OO?

- eine Art zu Programmieren, eine Modellierungstechnik?
- moderne Sprachen?
- OO = C++/Java?
- interpretierte Sprachen?
- OO = Klassen + Objekte?
- Sprachen mit Abstraktion und Kapselung?
- stark typisierte Sprachen?
- ...

Klassen und Objekte

- Klasse = Beschreibung von Objekten
- besteht aus Daten + Code
- Jargon: Felder, Attribute ... + Methoden
- oft Kapselung, insbesondere der Felder
- Was läßt sich mit Klassen machen?

⇒ genau 2 Dinge

- Instanzieren ⇒ neues Objekt
- Erweitern ⇒ neue Klasse¹

¹Vererbung kommt später

```
class Cell is
  var contents: int := 0;    // Feld

  method get (): int =      // Methode 1
    return contents;

  method set(x: int) =      // Methode 2
    contents := x;
end;

...

Cell x := new Cell();      // Instanziierung
```

Abbildung 1: Klassen und Objekte

Klassen = Module?

- Modul = Packet (“Struktur”) von zusammengehörigen Definitionen
- Daten- und Funktionsdefinitionen \Rightarrow Abstrakter Datentyp
- oft Möglichkeit der Kapselung
- konsistentes *Namensschema*
- \Rightarrow was also ist der Unterschied?

```
module IntList =
struct
  type intlist = int list    (* Implementierung des Typs *)

  let emptylist  = []        (* Seine Operationen *)
  let append l x = x::l
  let is_empty l  = match l with
    [] -> true
  | _  -> false

  let rec equals l1 l2 = match (l1, l2) with
    ([], []) -> true
  | (x1::l1, x2::l2) -> ((x1=x2) && (equals l1 l2))
  | _ -> false
end;;
```

Abbildung 2: Listen-ADT als Modul

Klasse = Typ?

- **Typ** = Bezeichnung der Menge seiner Werte/Programme
- charakterisiert, was man mit einem Programmstück **machen** darf
- \Rightarrow Nutzen: Testen zur Verhinderung von (bestimmten) **Laufzeitfehlern**
- Unterscheidung zwischen **statischer** und **dynamischen** Typen:

```
x: int;
```

```
if (4<5) then x:=5 else x:="abc";
```

Klasse = Typ? (Forts.)

- **Klassentypen** (*class types*) = Identifizierung von Klassennamen mit Typen

⇒ **Klassen** = instanziierbare Module und gleichzeitig Namen von Typen?

```
e: IntList.intlist = IntList.emptylist();;  
IntList.append(e, 3);;           // append ver"andere den Zustand
```

das selbe wie?

```
IntList e = new IntList();;  
e.append(3);;
```

Methodenaufruf?

Prozeduraufruf vs. Methodenaufruf?

- Jargon: **Methode** = Code, **Nachricht** (Message, Methodenselektor) = Name der Methode
- **Methodenaufruf** (*message passing, method lookup, method dispatch*)²
- wichtige Frage beim Methodenauf (z.B. `e.append(3)`), “wo” ist der Code, im Objekt oder in der Klasse/Typ³
- Antwort: egal, **solange** jede Objektvariable *genau* einer Klasse angehört/exakt einen Typ besitzt

²wg. C++ auch als **virtuelle** Methoden bezeichnet.

³das “wo” ist übertragen gemeint.

Polymorphie

Ein Programm aber mehrere Typen

- nicht speziell OO, aber wichtig dort
- unterschiedlicher Formen

Ad-hoc:	{	Überladen
	{	Coercion
universell (uniform)	{	parametrisch
	{	Inklusion, Subtyping

Untertypen

- **Untertypisierung**: **Ordnung** (\leq) auf den Typen
- Typ entspricht Menge seiner Elemente \Rightarrow Untertyp entspricht **Teilmenge**
- Typ spezifiziert **Verwendbarkeit** eines Objekts/Terms \Rightarrow Elemente des Untertyps können **anstelle** von solchen des Obertyps verwendet werden
- Jargon: “**is-a**”-Beziehung
- Form der **Polymorphie**
- Unterscheidung: **strukturell** vs. **deklarativ**

Untertypen

- Beispiel für eine strukturelle Regel

$$\frac{S_1 \leq S'_1 \quad T'_1 \leq T_1}{S'_1 \rightarrow T'_1 \leq S_1 \rightarrow T_1}$$

Vererbung

- wichtiges Konzept klassenbasierter OO-Sprache
- Vererbung:
 1. Vererbung als Widerspiegelung natürlicher Taxonomien (Typsicht der Klassen)
 2. Wiederverwendung von Code, inkrementelle Programmentwicklung

Vererbung und Untertypen

- oft Gleichsetzung von *Vererbung* und *Untertypen*
- *Grund*: oft
 - Gleichsetzung von *Klassen* und *Typen*, bzw. Objekttyp = Klasse
 - Vererbung: *einzig*e Weise, um Untertypen zu bekommen
- aber: *Vorsicht* wenn man (statische) *Typsicherheit* will

Vererbung und Untertypen? (2)

- **Klassentypen** (class types): Klassen sind Namen für Typen
- ⇒ **Vererbung** impliziert (vorteilhafterweise) **Untertypisierung**
- ⇒ Durch Vererbung werden nur neue Methoden **hinzugefügt**
- **Klassenhierarchie**, oft als **Baum**
- **Trennung** z.B. in Java und Modula-3: Klassen und **Interfaces**
- Jargon: **Interface**, **Objektprotokoll**
- Eine Klasse kann **mehrere** Interfaces **Implementieren**, aber: \neq **Mehrfachvererbung**
- `implements` = **deklarierte Untertyprelation**

```
interface PInterface {
    public int getx();
    public void setx(int x);
};

interface CInterface extends PInterface{ // erweitern!
    public Color getc();
    public void setc(Color x);
};

class Point implements PInterface {
    int x;
    public int getx() {return x;}
    public void setx(int y) {x = y;}
}

class CPoint extends Point implements CInterface { .....};
```

Überschreiben von Methoden

- nicht nur Erben von Methoden, auch Umdefinieren, Überschreiben (*overriding*)
- Oft in Zusammenhang mit abstrakten Klassen oder Methoden

```
class Cell {  
    int cont = 0;  
    int get() { return cont;}  
    void set(int x) { cont = x;}  
}
```

```
class RCell extends cell{  
    int backup = 0;  
    void set(int x) { backup = cont; cont = x;}  
    void restore() { cont = backup;}  
}
```

Späte Bindung

- Bindung = Zuordnung von Namen zu Kode
- interessant hier: von Nachrichten zu Methoden
- Jargon: **späte/dynamische** Bindung von Methoden, (*late binding, dynamic dispatch/lookup, message passing*)
- das Kriterium für OO-Sprachen
- **statische** Methode: abhängig von der statischen Klasse/Typ⁴

⁴static in Java

```
class A {
    public int x = 3;
    public void test () { System.out.println("A"); }
    public void test2 () { System.out.println("x = " + x); }
};

class B extends A {
    public int x = 4;
    public void test () { System.out.println("B"); }
    public void test2 () { System.out.println("x = " + x); }
};

public class Main {
    public static void main (String[] args) {
        A a = new A();
        A b; b = new B();
        b.test();
        b.test2();
        System.out.println("b.x = " + b.x);
    }
}
```

Objektidentität und `self`

- Objekte besitzen eine Identität
- können sich selbst referenzieren: `self`⁵
- natürlich `dynamisch`⁶
- das erste Argument aller Methoden, `implizit`

⁵Manchmal auch `this`

⁶statische Methoden/Klassenmethoden besitzen kein `self`.

Klassen und Module (2)

```
module type INTLIST =
sig
  type intlist
  val emptylist    : intlist
  val append       : intlist -> int -> intlist
  val is_empty     : intlist -> bool
  val equals       : intlist -> intlist -> bool
end;;
```

```
module IntList =
struct
  type intlist = int list
  let emptylist = []
  let append self x = x::self
  let is_empty self = match self with
    [] -> true
  | _  -> false.....
end;;
```

Überladen

- Überladen (*overloading*)

ein Name, unterschiedlicher Kode

- Form der Polymorphie (Jargon: Ad-Hoc-Polymorphie)
- drei Stufen der Komplexität:
 - eingebaut
 - statisch, benutzerdefiniert
 - dynamisch
- oft in OO: parametrisches Überladen⁷

⁷Beachte parametrisches Überladen \neq parametrische Polymorphie (= gekennzeichnet durch einen uniformen Kode für unterschiedliche Daten).

Message passing als dynamisches Überladen

- Methodenaufruf: interpretierbar als **Überladen** des Methodennamens

$$o.method(x) = method(o, x)^8$$

- Code: abhängig vom **dynamischen** Typ/Klasse des⁹ **ersten** Argumentes

⇒ **single Dispatch**-Sprachen (fast alle)

- **multiple Dispatch**: Verallgemeinerung auf alle Argumente (v.a. CLOS)

⁸Jargon: *pre-methods*

⁹impliziten

```
class A{
    public int x = 3;
    public boolean eqq(A a) { return (3 == a.x);};
}

class B extends A{
    public int x = 5;
    public int y = 6;
    public boolean eqq(B b) {return ((x == b.x) && (y == b.y));};
}

public class Binary { //main test program
    public static void main (String[] args) {
        A a = new A();
        B b = new B();
        if (a.eqq(a)) System.out.println(" gleich");
        else          System.out.println("ungleich");
    };};
```

Methodenspezialisierung

```
Class A {  
    T1 m1(x: S1) { .....};  
    T2 m2(x: S2) { .....};  
};
```

```
Class B extends A {  
    T1' m1(x: S1') {.....}  
};
```

Methodspezialisierung (2)

- da $B \leq A$, muß (für $s_1 : S_1$) erlaubt sein:

b.m1(s1)

- Sicherheit beim Überschreiben:
 - $T'_1 \leq T_1$, Kovariant¹⁰ und
 - $S_1 \leq S'_1$ kontravariant
- Jargon: Methodenspezialisierung durch Überschreiben
- Vergleiche: strukturelle \leq -Regel für Funktionen:

$$\frac{S_1 \leq S'_1 \quad T'_1 \leq T_1}{S'_1 \rightarrow T'_1 \leq S_1 \rightarrow T_1}$$

¹⁰bekannte Ausnahme: Eiffel: kovariant im Argument!

Methodenspezialisierung (3)

- durch Vererbung
- der "Typ" von `self` verändert sich kovariant
- implizite Spezialisierung
- Beachte den Unterschied von Vererbung und Überschreiben

Spezialisierung des Self-Typs

- Self: **Notation** für den dynamischen Typ des Objektes
 - Verhindert **Informationsverlust**
-

```
public class Dummy {  
    public Self clone() {  
        return new Self();  
    };};
```

- Jargon: “**run-time type identification**”
- `instance_of`, `type-case`

- korrekt unter der Annahme: Self steht dynamisch für **speziellere** Typen/Klassen¹¹
- Eiffel: erlaubt Self-Spezialierung auch im **Argumentposition**....

¹¹Vergleiche Methodenspezialisierung durch Überschreiben.

Literatur

- [Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Monographs in Computer Science. Springer.
- [Bruce et al., 1996a] Bruce, K., Cardelli, L., and Pierce, B. (1996a). Comparing object encodings. In Bruce, K., editor, *Informal Proceedings of the Third International Workshop on Foundations of Object-Oriented Languages (FOOL'96)*. Summary.
- [Bruce et al., 1996b] Bruce, K. B., Cardelli, L., Castagna, G., the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Leavens, G. T., and Pierce, B. (1996b). On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242.
- [Budd, 1997] Budd, T. (1997). *An Introduction to Object-Oriented Programming*. Addison-Wesley, 2 edition.
- [Castagna, 1997] Castagna, G. (1997). *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser.
- [Cook, 1991] Cook, W. (1991). Object-Oriented Programming Versus Abstract Data Types. In de Bakker, J. W., de Roever, W.-P., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages (REX Workshop)*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag.