Objektorientierte Entwurfsmus
in der Funktionalen Programmier

Johannes Waldmann (HTWK Leipzig)

Kolloquium HTWK Leipzig FB IMN, 14. Ju

Plan

- Abstraktionen
- Funktionales Programmieren
- Objektorientierte Entwurfsmuster
- Vergleich
- Folgerungen/Diskussion

Abstraktionen

Oberfläche eines Würfels mit Kantenlänge 4:
`return 6 * 4 * 4;`
...mit Kantenlänge 5: `return 6 * 5 * 5;`
`double f (double a) { return 6 * a * a;`

- Unterprogramm entsteht durch Abstraktion
  konkretem Programm(-Abschnitt):
  konkrete Werte werden durch formale Para
  (Platzhalter) ersetzt.
- bei Unterprogramm-Aufruf werden formale
  Parameter durch tatsächliche Werte ersetzt
  konkrete Instanz dieser Abstraktion erzeugt

Abstraktionen (II)

für Anweisungen: $\Rightarrow$ Blöcke, Unterprogramme
für Daten: oft sollen auch *Muster für Daten* vera
werden (weil es umständlich oder sogar unmög
alle Daten einzugeben)

- Wildcards bei Arbeit mit Dateisystem/Shell
- Gruppen für Rechtevergabe (`.htaccess`, `/etc/apache/httpd.conf`)

man *erkennt* und *benennt* Muster, um diese da
mehrmals zu *benutzen*. Jede Benutzung gesch
parametrisiert.
eine Abstraktion ist damit eine *Funktion*

Ein Abstraktions-Kalkül

das syntaktische Umformen von
Funktions-Definitionen und -Aufrufen beschreib
Lambda-Kalkül (Alonzo Church, *The Calculi of*
Lambda Conversion, 1936;
`http://www-groups.dcs.st-and.ac.uk/~hist`
Henk Barendregt, *Lambda Calculus—Syntax a*
Semantics, 1980; `http://www.cs.ru.nl/~henk`

```
S = \ x y z -> x z (y z)
I = \ x -> x ; K = \ x y -> x
    S K K z -> K z (K z) -> z
    S I I (S I I) -> I (S I I) (I (S I I
        -> S I I (S I I) -> ..
```

Lambda-Kalkül

Der Kalkül hört natürlich nicht bei Funktionen ü
Daten auf, sondern untersucht Funktionen von
Funktionen von Daten, Funktionen von . . . Date
Soweit nach „oben“ trauen sich (auch heute) nu
wenige Programmierer und Programmiersprach
Immerhin wird dieser Mangel immer stärker erk

Tatsächlich gibt es (in der reinen Variante) *nur*
Funktionen, und man „simuliert“ damit Daten (z
Zahlen)

Abstraktionen in Programmierspra

- wovon/was kann man abstrahieren/parametrisieren?
- kann man mit den Abstraktionen rechnen?
- C: Anweisungen → Unterprogramme (nicht dürfen Argument und Resultat von Unterprogrammen sein und in Datenstruktu vorkommen.
- Pascal: Anweisungen → Unterprogramme. lokal sein, Argument sein, aber nicht Resul
- C++/Java: Klassen und Unterprogramme — Templates/Generics

wenn bestimmte Abstraktionen nicht möglich si aber ihr Bedarf (mehr oder weniger diffus) erka wird, führt das zu Basteleien, die evtl. nach ein Weile kanonisiert werden (durch Sekundärlitera Werkzeuge).

Algebraische Datentypen

deklarieren:

```
data Tree a = Leaf
            | Node { left  :: Tree a
                   , key   :: a
                   , right :: Tree a
                   }
```

benutzen:

```
fib :: Int -> Tree Int
fib n = if n > 1
    then Node { left  = fib (n-1)
              , key   = n
              , right = fib (n-2)
              }
    else Leaf
```

Beispiel:

```
fib 3 = Node { left = Node {left = Leaf,
             , key = 3, right = Leaf
             }
```

Daten in Java

```
data Tree a = Leaf
  | Node { left  :: Tree a , key   :: a , right :: T
```

korrekte Übersetzung:

```
interface Tree<A> { }

class Leaf<A> implements Tree<A> { }
class Node<A> implements Tree<A> {
  Tree<A> left; A key; Tree<A> right;
  public Node(Tree<A> left, A key, Tree<A
    this.left = left; this.key = key;
    this.right = right;
  }
}
```

Zu einfach

```
class Tree<A> {
  Tree<A> left; A key; Tree<A> right;
  Tree<A> leaf() { return null; }
  Tree<A> node(Tree<A> left, A key, Tree<A> right)
    { return new Tree (left, key, right); }
  Tree (Tree<A> left, A key, Tree<A> right)
    { this.left = left; this.key = key; this.right =
}
class Tree<A> { ...
  int size () {
    return this.left.size()
       + 1 + this.right.size();
  }
}
```

Funktionen über Daten

Anzahl aller Nodes:

```
size :: Tree a -> Int
size t = case t of
    Leaf -> 0
    Node { } ->
        size (left t) + 1 + size (right t
```

Lister aller Schlüssel in Inorder-Reihenfolge:

```
inorder :: Tree a -> [a]
inorder t = case t of
    Leaf -> []
    Node { } -> inorder (left t)
       ++ [ key t ] ++ inorder (right t)
```

Rekursions-Schema

```
tfold :: b -> ( b -> a -> b -> b )
      -> ( Tree a -> b )
tfold leaf node = f where
    f t = case t of
        Leaf -> leaf
        Node { } -> node
            (f (left t)) (key t) (f (righ
```

ersetze jeden Konstruktor durch eine Funktion passender Stelligkeit (Katamorphismus)

Anwendungen:

```
size'    = tfold 0  ( \ l k r -> l  +  1
inorder' = tfold [] ( \ l k r -> l ++ [k
```

Homomorphismen

```
tmap :: (a -> b) -> ( Tree a -> Tree b )
tmap f t = case t of
    Leaf -> Leaf
    Node { } ->
        Node { left = tmap f (left t))
             , key = f (key t)
             , right = tmap f (right t)
             }
```

```
fib 3 =
Node {left = Node {left = Leaf, key = 2, right = Lea
tmap ( \ x -> x*x ) ( fib 3 ) =
Node {left = Node {left = Leaf, key = 4, right = Lea
```

Listen

```
data [a] -- Typname
   = []  -- Konstruktor für leere Liste
   | (:) { head :: a, tail :: [a] } -- (

length :: [a] -> Int
length l = case l of
    [] -> 0 ;     x : xs -> 1 + length x
```

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys ; (x : xs) ++ ys = x : (ys
```

Rekursionsschema?

Verifikation von Programmeigenscha

Beispiel: `size t == length ( inorder t )`

```
falls t == Leaf
    size Leaf =?= length ( inorder Leaf
    0         =?= length []
falls t = Node { left = l, key = k, right
    size t = size l + 1 + size r
        =?= length ( inorder l ++ [ k ]
        = length (inorder l) ++ length
```

benutzt Induktion nach `t` und ...

Verifikation von Programmeigenscha

```
length (xs ++ ys) = length xs + length y

Beweis:
length ( [] ++ ys ) = length ys = 0 + le

length ( (x : xs) ++ ys )
  = length (x : (xs ++ ys))
= 1 + length (xs ++ ys)
= 1 + (length xs + length ys)
= (1 + length xs) + length ys
= length (x : xs) + length ys
```

Suchbäume

`s` ist Suchbaum, wenn für jeden Teilbaum `t /=` von `s` gilt: für jeden Teilbaum `u /= Leaf` von `le` gilt `key u <= key t` und für jeden Teilbaum `v /` von `right t` gilt `key t <= key v`.
Ein Suchbaum repräsentiert eine Multimenge v Schlüsseln.
Die Inorder-Reihenfolge der Knoten eines Suchbaumes ist aufsteigend geordnet.

Funktionen für Suchbäume (I

```
contains :: Ord a => Tree a -> a -> Bool
contains t x = case t of
    Leaf -> False
    Node { } -> case compare x key of
        LT -> contains (left  t) x
        EQ -> True
        GT -> contains (right t) x
```

besser natürlich so:

```
contains t x =
    tfold ( False )
          ( \ l k r -> case compare x k
              LT -> l ; EQ -> True ; GT
          t
```

Funktionen für Suchbäume (I

```
build :: Ord a => [a] -> Tree a
build [] = Leaf
build (x : xs) =
    let ( low, high ) = partition ( < x
    in Node { left = build low
            , key = x
            , right = build high
            }
```

benutzt

```
partition ::
    ( a -> Bool ) -> [a] -> ( [a], [a] )
```

Suchbäume zum Sortieren

```
tsort :: Ord a => [a] -> [a]
tsort xs = inorder ( build xs )
```

Das ist ein (unter anderem Namen) sehr bekan
Sortierverfahren. Welches? Antwort durch
Programmtransformation.

Programm-Transformationen (

```
tsort [] = inorder ( build [] )
  = inorder Leaf = []
tsort (x : xs) = inorder ( build ( x : xs ))
  = inorder ( let ( low, high ) = partition (< x) xs
                in  Node { left = build low
                         , key = x
                         , right = build high
                         } )
= let ( low, high ) = partition (< x) xs
  in  inorder ( build low ) ++ [x] ++ inorder ( bu
= let ( low, high ) = partition (< x) xs
  in  tsort low ++ [x] ++ tsort high
```

Programm-Transformationen (

Resultat der Umformungen:

```
tsort [] = []
tsort (x : xs) =
    let ( low, high ) = partition ( < x
    in  tsort low ++ [x] ++ tsort high
```

alle `Tree`-Benutzungen sind eliminiert $\Rightarrow$ virtuel
Datenstrukturen.

Compiler kann solche Transformationen selbst
ausführen
(*equational reasoning* genügt, kein Hoare-Kalk
nötig, weil es keinen Programmzustand gibt).

Transformationen (III)

Transformationen werden unterstützt durch
Verwendung von Rekursions-Schemata (anstel
direkter Rekursion):
Compiler kann Recheneregeln benutzen, z. B.

```
tmap ( f . g ) = tmap f . tmap g

s f = f'  und  s (g l k r) = g' (s l) k
==> s . tfold f g = tfold f' g'

Bsp: length . inorder = size
```

Entwurfsmuster (I)

Erich Gamma, Richard Helm, Ralph Johnson, J
Vlissides: *Entwurfsmuster (design patterns)* —
Elemente wiederverwendbarer objektorientierte
Software, Addison-Wesley 1994.
liefert Muster (Vorlagen) für Gestaltung von
Beziehungen zwischen (mehreren) Klassen un
Objekten, die sich in wiederverwendbarer, flexi
Software bewährt haben.

Anregung: Christopher Alexander: *The Timeles*
of Building, A Pattern Language, Oxford Univ. P
1979.

Entwurfsmuster (II)

Jedes Muster beschreibt eine in unserer Umwe
beständig wiederkehrende Aufgabe und erläute
Kern ihrer Lösung. Wir können die Lösung beli
anwenden, aber niemals identisch wiederholen

Ausgangspunkt/Beispiel: Model/View/Controlle
Benutzerschnittstellen in Smalltalk-80)

- Aktualisierung von entkoppelten Objekten: *Beobachter*
- hierarchische Zusammensetzung von View-Objekten: *Komposition*
- Beziehung View–Controller: *Strategie*

Entwurfsmuster: Beispiele

Beobachter-Muster:

- Subjekt: Beobachter anmelden, abmelden
- Beobachter: aktualisieren

Besucher-Muster:

- Besucher: Aktion
- Element: Besucher empfangen
- Behälter: Besucher an alle Elemente

Iterator-Muster:

- Behälter: Iterator erzeugen
- Iterator: start, weiter, aktuelles, beendet

Musterkatalog

- Erzeugungsmuster:
  - klassenbasiert: Fabrikmethode
  - objektbasiert: abstrakte Fabrik, Erbauer, Prototyp, Singleton
- Strukturmuster:
  - klassenbasiert: Adapter
  - objektbasiert: Adapter, Brücke, Dekorier Fassade, Fliegengewicht, Kompositum,
- Verhaltensmuster:
  - klassenbasiert: Interpreter, Schablonenmethode
  - objektbasiert: Befehl, Beobachter, Besuc Iterator, Memento, Strategie, Vermittler, Zustand, Zuständigkeitskette

Wie Entwurfsmuster Probleme lö

- Finden passender Objekte
  insbesondere: nicht offensichtliche Abstrakt
- Bestimmen der Objektgranularität
- Spezifizieren von Objektschnittstellen und Objektimplementierungen
  unterscheide zwischen *Klasse* (konkreter T
  Typ (abstrakter Typ).
  programmiere auf eine Schnittstelle hin, nic
  eine Implementierung!
- Wiederverwendungsmechanismen anwend
  ziehe Objektkomposition der Klassenvererb
  vor
- Unterscheide zw. Übersetzungs- und Laufz

Muster anstatt Funktionen

einige Muster simulieren Funktionen als *first cl*
objects.
(In Java kann ein Unterprogramm nicht Argume
oder Resultat eines Unterprogramms sein, nich
Attribut eines Objektes.)
Typische Anwendungen:
Befehl (z. B. ActionListener),
Strategie (z. B. Comparator).
in beiden Fällen soll nur ein Unterprogramm als
Argument einer Methode erscheinen.
man muß stattdessen eine Klasse anlegen, die
gewünschte Methode enthält, und dann ein Ob
dieser Klasse als Argument nehmen.

Muster anstatt Funktionen (I

Muster *Interpreter*, Bestandteile: Ausdruck (mit
Methode: interpretiere), Kontext;
tatsächlich wird damit jedem Ausdruck ein
Unterprogramm zugeordnet (das einen Wert
berechnet oder den globalen Zustand ändert)
z. B. ist `tfold 0 (\ l k r -> l + 1 + r)` ein
Interpreter (ohne Kontext)

Muster anstatt Funktionen höherer O

einige Muster simulieren (die in der meisten OO-Sprachen fehlenden) Funktionen höherer Ordnung.
Iterator, Befehl:

```
map :: ( a -> b ) -> ( [a] -> [b] )
map f []       = []
map f (x : xs) = f x : map f xs
```

```
map ( \ x -> 3 * x ) [ 1, 2, 5 ] == [ 3,
```

Muster anstatt Konstruktorklas

In C++/Java gibt es Klassen-Schablonen, diese
können aber nicht frei verwendet werden (eine
Schablone kann kein Argument einer Schablon
In Haskell ist das Typklassensystem auch auf (
instantiierte) Typschablonen anwendbar

```
class Functor c where
    fmap :: (a -> b) -> ( c a -> c b )

map  :: ( a -> b ) -> ( [a]    ->    [b]
tmap :: ( a -> b ) -> ( Tree a -> Tree b

instance Functor List where fmap = map
instance Functor Tree where fmap = tmap
```

Zustands-Objekte

wiederverwendbare Komponenten dürfen *keine* Zustand enthalten. (vgl.: Unterprogramme dürfe keine globalen Variablen benutzen)
Zustand muß explizit gemacht werden (durch Zustands-Objekte)
in der (reinen) funktionalen Programmierung pa das von selbst: dort *gibt es keine Zuweisungen* const-Deklarationen mit einmaliger Initialisierun damit kann man Programmeigenschaften viel l beweisen (es gibt keinen versteckten Programmzustand).
alle Datenstrukturen sind automatisch persisten nichts zerstört/überschrieben werden kann.

Zusammenfassung (Thesen)

- Programmieren ist *Abstrahieren*, Parametri
- Der eigentliche Abstraktionskalkül ist der Lambda-Kalkül (mit *dependent types*).
- OO bietet („klassisch“) Klassen, und („richti Schnittstellen, d. h. abstrakte Datentypen.
- FP bietet Funktionen als gleichberechtigte
- moderne OO- und FP-Sprachen bieten gen Polymorphie (= Abstraktion über Typen)
- Abstraktionen *finden* und vernünftig *benutz*
- OO-Entwurfsmuster bieten dafür im Rahme OOP eine gute Orientierung.
- Viele Entwurfsprobleme haben aber eine kl FP-Lösung.